

Oracle® Fusion Middleware

Fusion Developer's Guide for Oracle Application Development
Framework

11g Release 1 (11.1.1)

B31974-02

November 2008

B31974-02

Copyright © 2008 Oracle. All rights reserved.

Primary Authors: Ralph Gordon (Lead), Peter Jew, Dave Mathews, Landon Ott, and Robin Whitmore

Contributing Authors: Walter Egan, Catherine Pickersgill, and Odile Sullivan-Tarazi

Contributors: Steve Muench, Lynn Munsinger

The Programs (which include both the software and documentation) contain proprietary information; they are provided under a license agreement containing restrictions on use and disclosure and are also protected by copyright, patent, and other intellectual and industrial property laws. Reverse engineering, disassembly, or decompilation of the Programs, except to the extent required to obtain interoperability with other independently created software or as specified by law, is prohibited.

The information contained in this document is subject to change without notice. If you find any problems in the documentation, please report them to us in writing. This document is not warranted to be error-free. Except as may be expressly permitted in your license agreement for these Programs, no part of these Programs may be reproduced or transmitted in any form or by any means, electronic or mechanical, for any purpose.

If the Programs are delivered to the United States Government or anyone licensing or using the Programs on behalf of the United States Government, the following notice is applicable:

U.S. GOVERNMENT RIGHTS Programs, software, databases, and related documentation and technical data delivered to U.S. Government customers are "commercial computer software" or "commercial technical data" pursuant to the applicable Federal Acquisition Regulation and agency-specific supplemental regulations. As such, use, duplication, disclosure, modification, and adaptation of the Programs, including documentation and technical data, shall be subject to the licensing restrictions set forth in the applicable Oracle license agreement, and, to the extent applicable, the additional rights set forth in FAR 52.227-19, Commercial Computer Software--Restricted Rights (June 1987). Oracle USA, Inc., 500 Oracle Parkway, Redwood City, CA 94065.

The Programs are not intended for use in any nuclear, aviation, mass transit, medical, or other inherently dangerous applications. It shall be the licensee's responsibility to take all appropriate fail-safe, backup, redundancy and other measures to ensure the safe use of such applications if the Programs are used for such purposes, and we disclaim liability for any damages caused by such use of the Programs.

Oracle, JD Edwards, PeopleSoft, and Siebel are registered trademarks of Oracle Corporation and/or its affiliates. Other names may be trademarks of their respective owners.

The Programs may provide links to Web sites and access to content, products, and services from third parties. Oracle is not responsible for the availability of, or any content provided on, third-party Web sites. You bear all risks associated with the use of such content. If you choose to purchase any products or services from a third party, the relationship is directly between you and the third party. Oracle is not responsible for: (a) the quality of third-party products or services; or (b) fulfilling any of the terms of the agreement with the third party, including delivery of products or services and warranty obligations related to purchased products or services. Oracle is not responsible for any loss or damage of any sort that you may incur from dealing with any third party.

Contents

Preface.....	xli
Audience.....	xli
Documentation Accessibility	xli
Related Documents	xlii
Conventions	xlii

Part I Getting Started with Fusion Web Applications

1 Introduction to Building Fusion Web Applications with Oracle ADF

1.1 Introduction to Oracle ADF.....	1-1
1.2 Oracle ADF Architecture	1-2
1.2.1 ADF Business Components.....	1-3
1.2.2 ADF Model Layer	1-4
1.2.3 ADF Controller.....	1-5
1.2.4 ADF Faces Rich Client.....	1-5
1.3 Developing Declaratively with Oracle ADF	1-6
1.3.1 Creating an Application Workspace.....	1-6
1.3.2 Creating Use Cases	1-9
1.3.3 Designing Application Control and Navigation Using ADF Task Flows.....	1-9
1.3.4 Identifying Shared Resources	1-11
1.3.5 Creating ADF Business Components to Access Data	1-12
1.3.5.1 Creating a Layer of Business Domain Objects for Tables.....	1-12
1.3.5.2 Building the Business Services	1-13
1.3.5.3 Testing Business Services with the Business Component Browser	1-14
1.3.6 Implementing the User Interface with JSF	1-14
1.3.7 Data Binding with Oracle ADF Model Layer	1-15
1.3.8 Validation and Error Handling.....	1-18
1.3.9 Adding Security	1-19
1.3.10 Testing and Debugging the Web Client Application	1-19
1.3.11 Refactoring Application Artifacts.....	1-19
1.3.12 Deploying a Fusion Web Application	1-20
1.4 Working Productively in Teams.....	1-20
1.4.1 Enforcing Standards	1-21
1.4.2 Using a Source Control System	1-22
1.5 Learning Oracle ADF	1-23

2 Introduction to the ADF Sample Application

2.1	Introduction to the Oracle Fusion Order Demo	2-1
2.2	Setting Up the Fusion Order Demo Application.....	2-1
2.2.1	How to Download the Application Resources.....	2-2
2.2.2	How to Install the Fusion Order Demo Application.....	2-2
2.2.3	How to Run the Fusion Order Demo Application.....	2-3
2.3	Taking a Look at the Fusion Order Demo Application.....	2-5
2.3.1	Anonymous Browsing	2-7
2.3.1.1	Viewing Product Details.....	2-8
2.3.1.2	Browsing the Product Catalog.....	2-11
2.3.1.3	Searching for Products.....	2-12
2.3.2	The Login Process	2-15
2.3.3	The Ordering Process.....	2-16
2.3.4	The Customer Registration Process	2-20

Part II Building Your Business Services

3 Getting Started with ADF Business Components

3.1	Introduction to ADF Business Components	3-1
3.1.1	ADF Business Components Features.....	3-2
3.1.2	ADF Business Components Core Objects	3-3
3.2	Comparison to Familiar 4GL Tools	3-3
3.2.1	Familiar Concepts for Oracle Forms Developers	3-3
3.2.1.1	Similarities Between the Application Module and a "Headless" Form Module .	3-4
3.2.1.2	Similarities Between the Entity Object and a Forms Record Manager.....	3-4
3.2.1.3	Similarities Between the View Object and a Data Block.....	3-5
3.2.2	Familiar Concepts for PeopleTools Developers	3-5
3.2.2.1	Similarities Between the Application Module and a "Headless" Component....	3-5
3.2.2.2	Similarities Between the Entity Object and a Record Definition	3-6
3.2.2.3	Similarities Between the View Object and a Row Set.....	3-6
3.2.3	Familiar Concepts for Siebel Tools Developers.....	3-6
3.2.3.1	Similarities Between the entity Object and a Table Object	3-6
3.2.3.2	Similarities Between the View Object and a Business Component.....	3-7
3.2.3.3	Similarities Between the Application Module and a Business Object	3-7
3.2.4	Familiar Functionality for ADO.NET Developers	3-7
3.2.4.1	Similarities Between the Application Module and a Data Set.....	3-7
3.2.4.2	Similarities Between the Entity Object and a Data Adapter	3-7
3.2.4.3	Similarities Between the View Object and a Data Table	3-7
3.3	Overview of Design Time Facilities	3-8
3.3.1	Choosing a Connection, SQL Flavor, and Type Map	3-8
3.3.2	Creating New Components Using Wizards	3-9
3.3.3	Creating New Components Using the Context Menu	3-9
3.3.4	Editing Components Using the Component Overview Editor.....	3-10
3.3.5	Visualizing, Creating, and Editing Components Using UML Diagrams	3-10
3.3.6	Testing Application Modules Using the Business Component Browser	3-10
3.3.7	Refactoring Components	3-10

3.4	Overview of the UI-Aware Data Model	3-10
3.4.1	A More Generic Business Service Solution	3-11
3.4.2	Typical Scenarios for a UI-Aware Data Model.....	3-11
3.4.3	UI-Aware Data Model Support for Custom Code.....	3-12
3.5	Overview of the Implementation Architecture	3-12
3.5.1	Standard Java and XML.....	3-13
3.5.2	Application Server or Database Independence.....	3-13
3.5.3	Java EE Design Pattern Support	3-13
3.5.4	Source Code Organization	3-13
3.5.5	Package Naming Conventions.....	3-14
3.5.6	Metadata with Optional Custom Java Code.....	3-15
3.5.6.1	Example of an XML-Only Component.....	3-16
3.5.6.2	Example of a Component with Custom Java Class	3-16
3.5.7	Basic Data Types	3-17
3.5.8	Generic Versus Strongly-Typed APIs.....	3-18
3.5.9	Custom Interface Support for Client-Accessible Components	3-19
3.5.9.1	Framework Client Interfaces for Components.....	3-19
3.5.9.2	Custom Client Interfaces for Components	3-19
3.6	Overview of Groovy Support.....	3-20

4 Creating a Business Domain Layer Using Entity Objects

4.1	Introduction to Entity Objects.....	4-1
4.2	Creating Entity Objects and Associations	4-2
4.2.1	How to Create Multiple Entity Objects and Associations from Existing Tables.....	4-2
4.2.2	How to Create Single Entity Objects Using the Create Entity Wizard	4-4
4.2.3	What Happens When You Create Entity Objects and Associations from Existing Tables 4-5	
4.2.3.1	What Happens When Tables Have Foreign Key Relationships	4-5
4.2.3.2	What Happens When a Table Has No Primary Key	4-6
4.2.4	What Happens When You Create an Entity Object for a Synonym or View.....	4-6
4.2.5	How to Edit an Existing Entity Object or Association	4-6
4.2.6	How to Create Database Tables from Entity Objects	4-7
4.2.7	How to Synchronize an Entity with Changes to Its Database Table.....	4-7
4.2.7.1	Removing an Attribute Associated with a Dropped Column	4-7
4.2.7.2	Addressing a Data Type change in the Underlying Table	4-8
4.2.8	How to Store Data Pertaining to a Specific Point in Time	4-8
4.2.9	What Happens When You Create Effective Dated Entity Objects	4-9
4.2.10	What You May Need to Know About Creating Entities from Tables.....	4-10
4.3	Creating and Configuring Associations	4-10
4.3.1	How to Create an Association	4-10
4.3.2	What Happens When You Create an Association	4-12
4.3.3	How to Change Entity Association Accessor Names.....	4-13
4.3.4	How to Rename and Move Associations to a Different Package	4-13
4.3.5	What You May Need to Know About Using a Custom View Object in an Association... 4-14	
4.3.6	What You May Need to Know About Composition Associations	4-14
4.4	Creating an Entity Diagram for Your Business Layer	4-15

4.4.1	How to Create an Entity Diagram.....	4-16
4.4.2	What Happens When You Create an Entity Diagram	4-17
4.4.3	What You May Need to Know About the XML Component Descriptors	4-18
4.4.4	What You May Need to Know About Changing the Names of Components.....	4-18
4.5	Defining Property Sets	4-18
4.5.1	How to Define a Property Set	4-19
4.5.2	How to Apply a Property Set	4-19
4.6	Defining Attribute Control Hints for Entity Objects	4-20
4.6.1	How to Add Attribute Control Hints	4-20
4.6.2	What Happens When You Add Attribute Control Hints	4-21
4.7	Working with Resource Bundles	4-21
4.7.1	How to Set Message Bundle Options	4-22
4.7.2	How to Use Multiple Resource Bundles	4-23
4.7.3	How to Internationalize the Date Format	4-24
4.8	Defining Business Logic Groups	4-24
4.8.1	How to Create a Business Logic Group.....	4-25
4.8.2	How to Create a Business Logic Unit	4-25
4.8.3	How to Override Attributes in a Business Logic Unit	4-26
4.8.4	What Happens When You Create a Business Logic Group	4-26
4.8.5	What Happens at Runtime: Invoking a Business Logic Group	4-27
4.9	Configuring Runtime Behavior Declaratively	4-27
4.9.1	How to Configure Declarative Runtime Behavior.....	4-28
4.9.2	What Happens When You Configure Declarative Runtime Behavior	4-29
4.10	Setting Attribute Properties.....	4-29
4.10.1	How to Set Database and Java Data Types for an Entity Object Attribute	4-29
4.10.2	How to Indicate Data Type Length, Precision, and Scale.....	4-30
4.10.3	How to Control the Updatability of an Attribute	4-31
4.10.4	How to Make an Attribute Mandatory.....	4-31
4.10.5	How to Define the Primary Key for the Entity.....	4-31
4.10.6	How to Define a Static Default Value.....	4-31
4.10.7	How to Define a Default Value Using a Groovy Expression	4-32
4.10.8	What Happens When You Create a Default Value Using a Groovy expression....	4-32
4.10.9	How to Synchronize with Trigger-Assigned Values.....	4-32
4.10.10	How to Get Trigger-Assigned Primary Key Values from a Database Sequence....	4-33
4.10.11	How to Protect Against Losing Simultaneously Updated Data	4-34
4.10.12	How to Track Created and Modified Dates Using the History Column.....	4-34
4.10.13	How to Configure Composition Behavior	4-35
4.10.13.1	Orphan-Row Protection for New Composed Entities	4-35
4.10.13.2	Ordering of Changes Saved to the Database	4-35
4.10.13.3	Cascade Update of Composed Details from Refresh-On-Insert Primary Keys	4-35
4.10.13.4	Cascade Delete Support.....	4-36
4.10.13.5	Cascade Update of Foreign Key Attributes When Primary Key Changes.....	4-36
4.10.13.6	Locking of Composite Parent Entities	4-36
4.10.13.7	Updating of Composing Parent History Attributes	4-36
4.10.14	How to Set the Discriminator Attribute for Entity Object Inheritance Hierarchies	4-36
4.10.15	How to Define Alternate Key Values	4-37
4.10.16	What Happens When You Define Alternate Key Values	4-37

4.10.17	What You May Need to Know About Alternate Key Values.....	4-37
4.11	Working Programmatically with Entity Objects and Associations	4-38
4.11.1	How to Find an Entity Object by Primary Key	4-38
4.11.2	How to Access an Associated Entity Using the Accessor Attribute	4-39
4.11.3	How to Update or Remove an Existing Entity Row	4-40
4.11.4	How to Create a New Entity Row	4-41
4.11.5	Assigning the Primary Key Value Using an Oracle Sequence.....	4-42
4.12	Generating Custom Java Classes for an Entity Object.....	4-43
4.12.1	How to Generate Custom Classes	4-43
4.12.2	What Happens When You Generate Custom Classes.....	4-44
4.12.3	What Happens When You Generate Entity Attribute Accessors	4-44
4.12.4	How to Navigate to Custom Java Files.....	4-45
4.12.5	What You May Need to Know About Custom Java Classes.....	4-45
4.12.5.1	About the Framework Base Classes for an Entity Object	4-46
4.12.5.2	You Can Safely Add Code to the Custom Component File	4-46
4.12.5.3	Configuring Default Java Generation Preferences	4-46
4.12.5.4	Attribute Indexes and InvokeAccessor Generated Code	4-46
4.12.6	Programmatic Example for Comparison Using Custom Entity Classes	4-48
4.13	Adding Transient and Calculated Attributes to an Entity Object	4-51
4.13.1	How to Add a Transient Attribute.....	4-51
4.13.2	What Happens When You Add a Transient Attribute.....	4-52
4.13.3	How to Base a Transient Attribute On a Groovy Expression	4-52
4.13.4	What Happens When You Base a Transient Attribute on Groovy Expression	4-54
4.13.5	How to Add Java Code in the Entity Class to Perform Calculation	4-54

5 Defining SQL Queries Using View Objects

5.1	Introduction to View Objects	5-1
5.1.1	Overview of View Object Concepts	5-2
5.1.2	Runtime Features Unique to Entity-Based View Objects	5-3
5.2	Populating View Object Rows from a Single Database Table.....	5-4
5.2.1	How to Create an Entity-Based View Object.....	5-5
5.2.1.1	Creating an Entity-Based View Object from a Single Table	5-5
5.2.1.2	Creating a View Object with All the Attributes of an Entity Object	5-8
5.2.2	What Happens When You Create an Entity-Based View Object.....	5-9
5.2.3	How to Create an Expert Mode, Read-Only View Object	5-10
5.2.4	What Happens When You Create a Read-Only View Object	5-12
5.2.5	How to Edit a View Object.....	5-13
5.2.5.1	Overriding the Inherit Properties from Underlying Entity Object Attributes ..	5-13
5.2.5.2	Controlling the Length, Precision, and Scale of View Object Attributes	5-14
5.2.6	How to Show View Objects in a Business Components Diagram.....	5-15
5.3	Populating View Object Rows with Static Data	5-16
5.3.1	How to Create Static View Objects with Data You Enter	5-16
5.3.2	How to Create Static View Objects with Data You Import	5-17
5.3.3	What Happens When You Create a Static List View Object	5-18
5.3.4	Editing Static List View Objects.....	5-20
5.3.5	What You May Need to Know About Static List View Objects.....	5-20
5.4	Limiting View Object Rows Using Effective Date Ranges.....	5-20

5.4.1	How to Create an Date-Effective View Object	5-20
5.4.2	How to Create New View Rows Using Date-Effective View Objects.....	5-21
5.4.3	How to Update Date-Effective View Rows	5-22
5.4.4	How to Delete Date-Effective View Rows	5-22
5.4.5	What Happens When You Create a Date-Effective View Object	5-23
5.4.6	What You May Need to Know About Date-Effective View Objects and View LInks..... 5-24	
5.5	Working with Multiple Tables in Join Query Results	5-24
5.5.1	How to Create Joins for Entity-Based View Objects.....	5-25
5.5.2	How to Select Additional Attributes from Reference Entity Usages	5-28
5.5.3	How to Remove Unnecessary Key Attributes from Reference Entity Usages	5-29
5.5.4	How to Hide the Primary Key Attributes from Reference Entity Usages.....	5-29
5.5.5	How to Modify a Default Join Clause to Be Outer Join When Appropriate.....	5-30
5.5.6	What Happens When You Reference Entities in a View Object.....	5-30
5.5.7	How to Create Joins for Read-Only View Objects	5-31
5.5.8	How to Test the Join View	5-32
5.5.9	How to Use the Query Builder with Read-Only View Objects.....	5-32
5.5.10	What You May Need to Know About Join View Objects	5-33
5.6	Working with Multiple Tables in a Master-Detail Hierarchy	5-33
5.6.1	How to Create a Master-Detail Hierarchy for Read-Only View Objects.....	5-34
5.6.2	How to Create a Master-Detail Hierarchy for Entity-Based View Objects	5-36
5.6.3	What Happens When You Create Master-Detail Hierarchies Using View Links...	5-37
5.6.4	How to Enable Active Master-Detail Coordination in the Data Model	5-38
5.6.5	How to Test Master-Detail Coordination.....	5-40
5.6.6	How to Access the Detail Collection Using the View Link Accessor	5-40
5.6.6.1	Accessing Attributes of Row by Name	5-40
5.6.6.2	Programmatically Accessing a Detail Collection Using the View Link Accessor..... 5-41	
5.7	Working with View Objects in Declarative SQL Mode.....	5-41
5.7.1	How to Create SQL-Independent View Objects with Declarative SQL Mode	5-42
5.7.2	How to Filter Declarative SQL-Based View Objects When Table Joins Apply	5-45
5.7.3	How to Filter Master-Detail Related View Objects with Declarative SQL Mode ...	5-46
5.7.4	How to Force Attribute Queries for Declarative SQL Mode View Objects.....	5-47
5.7.5	What Happens When You Create a View Object in Declarative SQL Mode	5-48
5.7.6	What Happens at Runtime	5-50
5.7.7	What You May Need to Know About Overriding Declarative SQL Mode Defaults	
5.7.7	5-50	
5.7.8	What You May Need to Know About Working Programmatically with Declarative SQL Mode View Objects	5-51
5.8	Working with View Objects in Expert Mode.....	5-51
5.8.1	How to Customize SQL Statements in Expert Mode	5-52
5.8.2	How to Name Attributes in Expert Mode.....	5-52
5.8.3	What Happens When You Enable Expert Mode.....	5-52
5.8.4	What You May Need to Know About Expert Mode	5-53
5.8.4.1	Expert Mode Provides Limited Attribute Mapping Assistance	5-53
5.8.4.2	Expert Mode Drops Custom Edits	5-54
5.8.4.3	Expert Mode Ignores Changes to SQL Expressions	5-54

5.8.4.4	Expert Mode Returns Error for SQL Calculations that Change Entity Attributes..... 5-55	
5.8.4.5	Expert Mode Retains Formatting of SQL Statement	5-56
5.8.4.6	Expert Mode Wraps Queries as Inline Views.....	5-56
5.8.4.7	Limitation of Inline View Wrapping at Runtime.....	5-57
5.8.4.8	Expert Mode Changes May Affect Dependent Objects	5-57
5.9	Working with Bind Variables.....	5-58
5.9.1	How to Add Bind Variables to a View Object Definition	5-58
5.9.2	What Happens When You Add Named Bind Variables.....	5-60
5.9.3	How to Test Named Bind Variables	5-60
5.9.4	How to Add a WHERE Clause with Named Bind Variables at Runtime	5-61
5.9.5	How to Set Existing Bind Variable Values at Runtime	5-63
5.9.6	What Happens at Runtime	5-64
5.9.7	What You May Need to Know About Named Bind Variables	5-65
5.9.7.1	An Error Related to Clearing Bind Variables	5-65
5.9.7.2	A Helper Method to Remove Named Bind Variables	5-66
5.9.7.3	Errors Related to Naming Bind Variables.....	5-67
5.9.7.4	Default Value of NULL for Bind Variables.....	5-67
5.10	Working with Named View Criteria.....	5-67
5.10.1	How to Create Named View Criteria Declaratively	5-68
5.10.2	How to Set User Interface Hints on View Criteria.....	5-72
5.10.3	How to Test View Criteria Using the Business Component Browser.....	5-75
5.10.4	How to Create View Criteria Programmatically.....	5-76
5.10.5	What Happens at Runtime	5-77
5.10.6	What You May Need to Know About the View Criteria API	5-78
5.10.6.1	Referencing Attribute Names in View Criteria	5-78
5.10.6.2	Referencing Bind Variables in View Criteria.....	5-78
5.10.6.3	Altering Compound Search Conditions Using Multiple View Criteria Rows ..	5-79
5.10.6.4	Searching for a Row Whose Attribute Value Is NULL Value.....	5-79
5.10.6.5	Searching Case-Insensitively	5-79
5.10.6.6	Clearing View Criteria in Effect	5-79
5.10.7	What You May Need to Know About Query-By-Example Criteria.....	5-79
5.11	Working with List of Values (LOV) in View Object Attributes	5-80
5.11.1	How to Define a Single LOV-Enabled View Object Attribute	5-82
5.11.2	How to Define Cascading Lists for LOV-Enabled View Object Attributes	5-83
5.11.3	How to Set User Interface Hints on a View Object LOV-Enabled Attribute	5-86
5.11.4	How to Test LOV-Enabled Attributes Using the Business Component Browser ...	5-89
5.11.5	What Happens When You Define an LOV for a View Object Attribute	5-90
5.11.6	What Happens at Runtime: When an LOV Queries the List Data Source	5-91
5.11.7	What You May Need to Know About Lists	5-92
5.11.7.1	Inheritance of AttributeDef Properties from Parent View Object Attributes ...	5-92
5.11.7.2	Using Validators to Validate Attribute Values	5-92
5.12	Defining Attribute Control Hints for View Objects.....	5-93
5.12.1	How to Add Attribute Control Hints	5-93
5.12.2	What Happens When You Add Attribute Control Hints	5-93
5.12.3	What You May Need to Know About Resource Bundles.....	5-94
5.13	Adding Calculated and Transient Attributes to a View Object	5-95

5.13.1	How to Add a SQL-Calculated Attribute.....	5-95
5.13.2	What Happens When You Add a SQL-Calculated Attribute	5-96
5.13.3	How to Add a Transient Attribute.....	5-97
5.13.4	What Happens When You Add a Transient Attribute.....	5-99
5.13.5	Adding Java Code in the View Row Class to Perform Calculation	5-99
5.13.6	What You May Need to Know About Transient Attributes.....	5-100

6 Working with View Object Query Results

6.1	Introduction to View Object Runtime Behavior.....	6-1
6.2	Creating an Application Module to Test View Instances	6-1
6.3	Testing View Object Instances Using the Business Component Browser	6-4
6.3.1	How to Run the Business Component Browser.....	6-5
6.3.2	How to Test Entity-Based View Objects Interactively	6-6
6.3.3	What Happens When You Use the Business Component Browser	6-7
6.3.4	How to Simulate End-User Interaction in the Business Component Browser	6-8
6.3.4.1	Testing Master-Detail Coordination	6-10
6.3.4.2	Testing UI Control Hints	6-10
6.3.4.3	Testing Business Domain Layer Validation.....	6-10
6.3.4.4	Testing Alternate Language Message Bundles and Control Hints	6-10
6.3.4.5	Testing View Objects That Reference Entity Usages.....	6-11
6.3.4.6	Testing Row Creation and Default Value Generation	6-11
6.3.4.7	Testing That New Detail Rows Have Correct Foreign Keys.....	6-11
6.3.5	How to Test Multiuser Scenarios in the Business Component Browser	6-11
6.3.6	How to Customize Configuration Options Before Running the Browser.....	6-12
6.3.7	How to Enable ADF Business Components Debug Diagnostics	6-12
6.3.8	What Happens at Runtime: When View Objects and Entity Objects Cooperate	6-13
6.3.8.1	What Happens When a View Object Executes Its Query	6-13
6.3.8.2	What Happens When a View Row Attribute Is Modified.....	6-15
6.3.8.3	What Happens When a Foreign Key Attribute is Changed.....	6-16
6.3.8.4	What Happens When a Transaction is Committed.....	6-17
6.3.8.5	What Happens When a View Object Requeries Data	6-18
6.3.9	What You May Need to Know About Optimizing View Object Runtime Performance ..	
	6-19	
6.4	Testing View Object Instances Programmatically.....	6-21
6.4.1	ViewObject Interface Methods for Working with the View Object's Default RowSet.....	
	6-22	
6.4.1.1	The Role of the Key Object in a View Row or Entity Row	6-22
6.4.1.2	The Role of the Entity Cache in the Transaction	6-23
6.4.2	How to Create a Command-Line Java Test Client.....	6-25
6.4.3	What Happens When You Run a Test Client Program.....	6-27
6.4.4	What You May Need to Know About Running a Test Client.....	6-27
6.4.5	How to Count the Number of Rows in a Row Set.....	6-28
6.4.6	How to Access a Detail Collection Using the View Link Accessor	6-28
6.4.7	How to Iterate Over a Master-Detail-Detail Hierarchy.....	6-30
6.4.8	How to Find a Row and Update a Foreign Key Value.....	6-31
6.4.9	How to Create a New Order	6-33
6.4.10	How to Retrieve the Row Key Identifying a Row	6-34

7 Defining Validation and Business Rules Declaratively

7.1	Introduction to Declarative Validation.....	7-1
7.1.1	When to Use Business-Layer Validation or Model-Layer Validation.....	7-2
7.2	Understanding the Validation Cycle	7-2
7.2.1	Types of Entity Object Validation Rules	7-2
7.2.1.1	Attribute-Level Validation Rules	7-3
7.2.1.2	Entity-Level Validation Rules.....	7-3
7.2.2	Understanding Commit Processing and Validation	7-3
7.2.3	Understanding the Impact of Composition on Validation Order	7-4
7.2.4	Avoiding Infinite Validation Cycles	7-4
7.2.5	What Happens When Validations Fail.....	7-4
7.2.6	Understanding Entity Objects Row States	7-5
7.3	Adding Validation Rules to Entity Objects and Attributes	7-5
7.3.1	How to Add a Validation Rule to an Entity or Attribute	7-6
7.3.2	How to View and Edit a Validation Rule On an Entity or Attribute	7-6
7.3.3	What Happens When You Add a Validation Rule.....	7-6
7.3.4	What You May Need to Know About Entity and Attribute Validation Rules.....	7-7
7.4	Using the Built-in Declarative Validation Rules	7-8
7.4.1	How to Ensure That Key Values Are Unique.....	7-8
7.4.2	What Happens When You Use a Unique Key Validator	7-9
7.4.3	How to Validate Based on a Comparison	7-9
7.4.4	What Happens When You Validate Based on a Comparison.....	7-11
7.4.5	How to Validate Using a List of Values	7-12
7.4.6	What Happens When You Validate Using a List of Values	7-13
7.4.7	What You May Need to Know About the List Validator	7-13
7.4.8	How to Make Sure a Value Falls Within a Certain Range.....	7-14
7.4.9	What Happens When You Use a Range Validator	7-14
7.4.10	How to Validate Against a Number of Bytes or Characters	7-15
7.4.11	What Happens When You Validate Against a Number of Bytes or Characters	7-15
7.4.12	How to Validate Using a Regular Expression	7-16
7.4.13	What Happens When You Validate Using a Regular Expression.....	7-17
7.4.14	How to Use the Average, Count, or Sum to Validate a Collection	7-17
7.4.15	What Happens When You Use Collection Validation	7-18
7.4.16	How to Determine Whether a Key Exists	7-18
7.4.17	What Happens When You Use a Key Exists Validator.....	7-20
7.4.18	What You May Need to Know About Declarative Validators and View Accessors	7-20
7.5	Using Groovy Expressions For Validation and Business Rules.....	7-21
7.5.1	How to Reference Entity Object Methods in Groovy Validation Expressions	7-21
7.5.2	How to Validate Using a True/False Expression	7-22
7.5.3	What Happens When You Add a Groovy Expression.....	7-23
7.6	Triggering Validation Execution	7-25
7.6.1	How to Specify Which Attributes Fire Validation	7-25
7.6.2	How to Set Preconditions for Validation	7-25
7.6.3	How to Set Transaction-Level Validation	7-25
7.6.4	What You May Need to Know About the Order of Validation Execution	7-26
7.7	Creating Validation Error Messages	7-26

7.7.1	How to Create Validation Error Messages	7-26
7.7.2	How to Localize Validation Messages.....	7-27
7.7.3	How to Conditionally Raise Error Messages Using Groovy.....	7-27
7.7.4	How to Embed a Groovy Expression in an Error Message.....	7-28
7.8	Setting the Severity Level for Validation Exceptions	7-28
7.9	Bulk Validation in SQL	7-29

8 Implementing Validation and Business Rules Programmatically

8.1	Introduction to Programmatic Business Rules	8-1
8.2	Using Method Validators.....	8-2
8.2.1	How to Create an Attribute-Level Method Validator	8-3
8.2.2	What Happens When You Create an Attribute-Level Method Validator.....	8-4
8.2.3	How to Create an Entity-Level Method Validator.....	8-5
8.2.4	What Happens When You Create an Entity-Level Method Validator	8-6
8.2.5	What You May Need to Know About Translating Validation Rule Error Messages	8-6
8.3	Assigning Programmatically Derived Attribute Values.....	8-7
8.3.1	How to Provide Default Values for New Rows at Create Time	8-7
8.3.1.1	Choosing Between <code>create()</code> and <code>initDefaultExpressionAttributes()</code> Methods	8-7
8.3.1.2	Eagerly Defaulting an Attribute Value from a Database Sequence	8-7
8.3.2	How to Assign Derived Values Before Saving.....	8-8
8.3.3	How to Assign Derived Values When an Attribute Value is Set	8-9
8.4	Undoing Pending Changes to an Entity Using the Refresh Method	8-9
8.4.1	How to Control What Happens to New Rows During a Refresh	8-10
8.4.2	How to Cascade Refresh to Composed Children Entity Rows.....	8-10
8.5	Using View Objects for Validation	8-10
8.5.1	How to Use View Accessors for Validation Against View Objects.....	8-10
8.5.2	How to Validate Conditions Related to All Entities of a Given Type.....	8-11
8.6	Accessing Related Entity Rows Using Association Accessors	8-11
8.6.1	How to Access Related Entity Rows.....	8-11
8.6.2	How to Access Related Entity RowSets of Rows	8-12
8.7	Referencing Information About the Authenticated User.....	8-12
8.8	Accessing Original Attribute Values.....	8-13
8.9	Storing Information About the Current User Session	8-13
8.9.1	How to Store Information About the Current User Session	8-13
8.10	Accessing the Current Date and Time	8-14
8.11	Sending Notifications Upon a Successful Commit	8-14
8.12	Conditionally Preventing an Entity Row from Being Removed.....	8-14
8.13	Determining Conditional Updatability for Attributes	8-15

9 Implementing Business Services with Application Modules

9.1	Introduction to Application Modules	9-1
9.2	Creating and Modifying an Application Module	9-3
9.2.1	How to Create an Application Module	9-3
9.2.2	What Happens When You Create an Application Module	9-4
9.2.3	How to Add a View Object to an Application Module.....	9-4
9.2.4	What Happens When You Add a View Object to an Application Module	9-9
9.2.5	How to Edit an Existing Application Module	9-10

9.2.6	How to Change the Data Control Name Before You Begin Building Pages.....	9-10
9.2.7	What You May Need to Know About Application Module Granularity.....	9-11
9.2.8	What You May Need to Know About View Object Components and View Object Instances	9-11
9.3	Configuring Your Application Module Database Connection	9-12
9.3.1	How to Use a JDBC URL Connection Type.....	9-12
9.3.2	How to Use a JDBC Data Source Connection Type.....	9-12
9.3.3	How to Change Your Application Module's Runtime Configuration.....	9-13
9.3.4	How to Change the Database Connection for Your Project	9-14
9.3.5	What You May Need to Know About Application Module Connections	9-15
9.4	Defining Nested Application Modules.....	9-15
9.4.1	How to Define a Nested Application Module.....	9-16
9.4.2	What You May Need to Know About Root Application Modules Versus Nested Application Module Usages	9-17
9.5	Creating an Application Module Diagram for Your Business Service	9-17
9.5.1	How to Create an Application Module Diagram.....	9-17
9.5.2	What Happens When You Create an Application Module Diagram	9-18
9.5.3	What You May Need to Know About Application Module Diagrams	9-18
9.5.3.1	Using the Diagram for Editing the Application Module.....	9-18
9.5.3.2	Controlling Display Options.....	9-19
9.5.3.3	Filtering Method Names.....	9-19
9.5.3.4	Showing Related Objects and Implementation Files.....	9-20
9.5.3.5	Publishing the Application Module Diagram	9-21
9.5.3.6	Testing the Application Module from the Diagram.....	9-21
9.6	Supporting Multipage Units of Work	9-21
9.6.1	How to Simulate State Management in the Business Component Browser	9-21
9.6.2	What Happens When the Application Uses Application Module Pooling and State Management	9-22
9.7	Customizing an Application Module with Service Methods.....	9-23
9.7.1	How to Generate a Custom Class for an Application Module	9-23
9.7.2	What Happens When You Generate a Custom Class for an Application Module.	9-24
9.7.3	What You May Need to Know About Default Code Generation.....	9-24
9.7.4	How to Add a Custom Service Method to an Application Module.....	9-25
9.7.5	How to Test the Custom Application Module Using a Static Main Method.....	9-27
9.8	Publishing Custom Service Methods to UI Clients.....	9-28
9.8.1	How to Publish a Custom Method on the Application Module's Client Interface.	9-28
9.8.2	What Happens When You Publish Custom Service Methods	9-29
9.8.3	How to Generate Client Interfaces for View Objects and View Rows.....	9-30
9.8.4	What You May Need to Know About Method Signatures on the Client Interface	9-31
9.8.5	What You May Need to Know About Passing Information from the Data Model	9-31
9.9	Debugging the Application Module Using the Business Component Browser	9-32
9.10	Working Programmatically with an Application Module's Client Interface	9-33
9.10.1	How to Work Programmatically with an Application Module's Client Interface..	9-33
9.10.2	What Happens When You Work with an Application Module's Client Interface..	9-34
9.10.3	How to Access an Application Module Client Interface in a Fusion Web Application....	
	9-35	
9.11	Overriding Built-in Framework Methods	9-37
9.11.1	How to Override a Built-in Framework Method	9-37

9.11.2	What Happens When You Override a Built-in Framework Method	9-38
9.11.3	How to Override <code>prepareSession()</code> to Set Up an Application Module for a New User Session	9-39

10 Sharing Application Module View Instances

10.1	Introduction to Shared Application Modules.....	10-1
10.2	Sharing an Application Module Instance.....	10-1
10.2.1	How to Create a Shared Application Module Instance	10-2
10.2.2	What Happens When You Define a Shared Application Module.....	10-3
10.2.3	What You May Need to Know About Shared Application Module Instances.....	10-5
10.2.3.1	Design Time Scope of the Shared Application Module	10-5
10.2.3.2	Design Time Scope of View Instances of the Shared Application Module.....	10-5
10.3	Defining a Base View Object for Use with Lookup Tables	10-5
10.3.1	How to Create a Base View Object Definition for a Lookup Table	10-6
10.3.2	What Happens When You Create a Base View Object.....	10-7
10.3.3	How to Define the WHERE Clause of the Lookup View Object Using View Criteria.....	10-10
10.3.4	What Happens When You Create a View Criteria with the Editor.....	10-11
10.3.5	What Happens at Runtime When a View Instance Accesses Lookup Data.....	10-12
10.4	Accessing View Instances of the Shared Service.....	10-12
10.4.1	How to Create a View Accessor for an Entity Object or View Object.....	10-13
10.4.2	How to Validate Against a View Accessor	10-15
10.4.3	What Happens When You Validate Against a View Accessor	10-16
10.4.4	How to Create an LOV Based on a Lookup Table	10-17
10.4.5	What Happens When You Define an LOV for a View Object Attribute	10-18
10.4.6	How to Automatically Refresh the View Object of the View Accessor	10-19
10.4.7	What Happens at Runtime	10-20
10.4.8	What You May Need to Know About Lists	10-20
10.4.8.1	Inheritance of AttributeDef Properties from Parent View Object Attributes.	10-20
10.4.8.2	Using Validators to Validate Attribute Values	10-21
10.5	Testing View Object Instances in a Shared Application Module.....	10-21
10.5.1	How to Test the Base View Object Using the Business Component Browser	10-21
10.5.2	How to Test LOV-Enabled Attributes Using the Business Component Browser .	10-22
10.5.3	What Happens When You Use the Business Component Browser	10-23
10.5.4	What Happens at Runtime When Another Service Accesses the Shared Application Module Cache	10-23

11 Using Oracle ADF Model in a Fusion Web Application

11.1	Introduction to ADF Data Binding.....	11-1
11.2	Exposing Application Modules with Oracle ADF Data Controls.....	11-3
11.2.1	How an Application Module Data Control Appears in the Data Controls Panel...	11-4
11.2.1.1	How the Data Model and Service Methods Appear in the Data Controls Panel.....	11-6
11.2.1.2	How Transaction Control Operations Appear in the Data Controls Panel	11-6
11.2.1.3	How View Objects Appear in the Data Controls Panel	11-7
11.2.1.4	How Nested Application Modules Appear in the Data Controls Panel	11-9
11.2.2	How to Open the Data Controls Panel	11-10

11.2.3	How to Refresh the Data Control	11-10
11.2.4	Packaging a Data Control for Use in Another Project.....	11-11
11.3	Using the Data Controls Panel.....	11-11
11.3.1	How to Use the Data Controls Panel	11-14
11.3.2	What Happens When You Use the Data Controls Panel.....	11-15
11.3.3	What Happens at Runtime: How the Binding Context Works.....	11-17
11.4	Working with the DataBindings.cpx File	11-18
11.4.1	How JDeveloper Creates a DataBindings.cpx File	11-18
11.4.2	What Happens When JDeveloper Creates a DataBindings.cpx File	11-18
11.5	Configuring the ADF Binding Filter	11-21
11.5.1	How JDeveloper Configures the ADF Binding Filter.....	11-21
11.5.2	What Happens When JDeveloper Configures an ADF Binding Filter	11-21
11.5.3	What Happens at Runtime: How the ADF Binding Filter Works	11-21
11.6	Working with Page Definition Files	11-22
11.6.1	How JDeveloper Creates a Page Definition File	11-22
11.6.2	What Happens When JDeveloper Creates a Page Definition File	11-23
11.6.2.1	Bindings Binding Objects	11-26
11.6.2.2	Executable Binding Objects.....	11-27
11.7	Creating ADF Data Binding EL Expressions	11-30
11.7.1	How to Create an ADF Data Binding EL Expression.....	11-31
11.7.1.1	Opening the Expression Builder from the Property Inspector	11-32
11.7.1.2	Using the Expression Builder	11-32
11.7.2	What You May Need to Know About ADF Binding Properties.....	11-33
11.8	Using Simple UI First Development	11-33
11.8.1	How to Apply ADF Model Data Binding to Existing UI Components	11-35
11.8.2	What Happens When You Apply ADF Model Data Binding to UI Components	11-36

12 Integrating Web Services Into a Fusion Web Application

12.1	Introduction to Web Services in Fusion Web Applications.....	12-1
12.2	Calling a Web Service from an Application Module.....	12-2
12.2.1	How to Call an External Service Programmatically	12-2
12.2.1.1	Creating a Web Service Proxy Class to Programmatically Access the Service	12-3
12.2.1.2	Calling the Web Service Proxy Template to Invoke the Service.....	12-3
12.2.1.3	Calling a Web Service Method Using the Proxy Class in an Application Module....	
	12-3	
12.2.2	What Happens When You Create the Web Service Proxy	12-4
12.2.3	What Happens at Runtime: When You Call a Web Service Using a Web Service Proxy Class	12-5
12.2.4	What You May Need to Know About Web Service Proxies	12-5
12.2.4.1	Using a Try-Catch Block to Handle Web Service Exceptions	12-5
12.2.4.2	Separating Application Module and Web Services Transactions	12-6
12.2.4.3	Setting Browser Proxy Information	12-6
12.2.4.4	Invoking Application Modules with a Web Service Proxy Class.....	12-6
12.3	Creating Web Service Data Controls.....	12-6
12.3.1	How to Create a Web Service Data Control.....	12-7
12.3.2	How to Adjust the Endpoint for a Web Service Data Control.....	12-7
12.3.3	What You May Need to Know About Web Service Data Controls.....	12-7

12.4	Securing Web Service Data Controls	12-9
12.4.1	WS-Security Specification.....	12-10
12.4.2	How to Create and Use Key Stores.....	12-10
12.4.2.1	Creating a Key Store.....	12-11
12.4.2.2	Requesting a Certificate.....	12-12
12.4.2.3	Exporting a Public Key Certificate.....	12-13
12.4.3	How to Define Web Service Data Control Security	12-14
12.4.3.1	Setting Authentication	12-15
12.4.3.2	Setting Digital Signatures.....	12-16
12.4.3.3	Setting Encryption and Decryption	12-17
12.4.3.4	Using a Key Store	12-17

Part III Creating ADF Task Flows

13 Getting Started with ADF Task Flows

13.1	Introduction to ADF Task Flows	13-1
13.1.1	Task Flow Advantages.....	13-2
13.1.2	Task Flow Types	13-2
13.1.2.1	ADF Unbounded Task Flows	13-3
13.1.2.2	ADF Bounded Task Flows	13-4
13.1.3	Control Flows	13-7
13.2	Creating Task Flows	13-8
13.2.1	How to Create an ADF Task Flow	13-9
13.2.2	How to Add an Activity to an ADF Task Flow	13-10
13.2.3	How to Add Control Flows.....	13-12
13.2.4	How to Add a Wildcard Control Flow Rule	13-14
13.2.5	What Happens When You Create a Control Flow Rule.....	13-15
13.2.6	What Happens at Runtime	13-16
13.2.7	What Happens When You Create an ADF Task Flow	13-16
13.2.8	What Happens at Runtime: Using ADF Task Flows	13-17
13.2.9	What You May Need to Know About Managed Beans	13-18
13.2.10	What You May Need to Know About Memory Scopes	13-19
13.2.10.1	View Scope	13-20
13.2.10.2	Backing Bean Scope.....	13-20
13.3	Adding a Bounded Task Flow to a Page	13-20
13.4	Designating a Default Activity in an ADF Bounded Task Flow	13-21
13.5	Running ADF Task Flows	13-22
13.5.1	How to Run a Task Flow Definition That Contains Pages	13-22
13.5.2	How to Run a Task Flow Definition That Uses Page Fragments	13-22
13.5.3	How to Run a Task Flow Definition That Has Parameters	13-23
13.5.4	How to Run a JSF Page	13-23
13.5.5	How to Run an ADF Unbounded Task Flow	13-23
13.5.6	How to Set a Run Configuration for a Project	13-24
13.6	Setting Project Properties for ADF Task Flows	13-25
13.7	Refactoring to Create New ADF Task Flows and Templates.....	13-25
13.7.1	How to Create an ADF Bounded Task Flow from Selected Activities	13-25
13.7.2	How to Create a Task Flow from JSF Pages.....	13-26

13.7.3	How to Convert ADF Bounded Task Flows	13-26
13.8	What You Should Know About Task Flow Constraints	13-27

14 Working with Task Flow Activities

14.1	Introduction to Activity Types.....	14-1
14.2	Using View Activities.....	14-3
14.2.1	Adding a View Activity	14-4
14.2.2	Transitioning Between View Activities	14-4
14.2.2.1	How to Transition to a View Activity	14-5
14.2.2.2	What Happens When You Transition Between Activities	14-6
14.2.3	Bookmarking View Activities	14-6
14.2.3.1	How to Create a Bookmarkable View Activity.....	14-7
14.2.3.2	How to Specify HTTP Redirect	14-8
14.2.3.3	What Happens When You Designate a View as Bookmarkable	14-9
14.3	Using URL View Activities.....	14-9
14.3.1	Constructing a URL for Use Within a Portlet	14-10
14.4	Using Router Activities.....	14-11
14.5	Using Method Call Activities	14-13
14.5.1	How to Add a Method Call Activity	14-14
14.5.2	How to Specify Method Parameters and Return Values	14-17
14.5.3	What Happens When You Add a Method Call Activity	14-17
14.6	Using Task Flow Call Activities.....	14-18
14.6.1	How to Call an ADF Bounded Task Flow	14-18
14.6.2	How to Specify Input Parameters on a Task Flow Call Activity	14-19
14.6.3	How to Call an ADF Bounded Task Flow Located in Another Web Application	14-20
14.6.4	How to Call a Bounded Task Flow with a URL	14-21
14.6.5	Specifying a Parameter Converter	14-22
14.6.6	How to Specify Before and After Listeners.....	14-22
14.6.7	What Happens When You Add a Task Flow Call Activity	14-23
14.6.8	What Happens at Runtime: Using a Task Flow Call Activity	14-23
14.7	Using Task Flow Return Activities	14-24
14.8	Using Save Point Restore Activities	14-26
14.9	Using Parent Action Activities.....	14-26

15 Using Parameters in Task Flows

15.1	Passing Parameters to a View Activity	15-1
15.2	Passing Parameters to an ADF Bounded Task Flow	15-2
15.3	Sharing Data Control Instances	15-6
15.3.1	What You May Need to Know About Managing Transactions.....	15-7
15.4	Specifying Return Values	15-8
15.5	Specifying EL Binding Expressions.....	15-9

16 Using ADF Task Flows as Regions

16.1	Introduction to ADF Regions	16-1
16.1.1	How to Create an ADF Region	16-3
16.1.2	How to Add a Task Flow Binding Declaratively	16-6

16.1.3	How to Specify Parameters for an ADF Region.....	16-6
16.1.4	What Happens When You Create an ADF Region	16-8
16.1.5	What Happens at Runtime: Executing an ADF Region.....	16-9
16.1.6	How to Trigger Navigation of an ADF Region's Parent Task Flow	16-9
16.1.7	What You May Need to Know About Refreshing an ADF Region.....	16-10
16.1.8	What You May Need to Know About Returning from an ADF Region	16-11
16.1.9	What You May Need to Know About Page Fragments.....	16-11
16.1.10	What You May Need to Know about ViewPorts	16-12
16.1.11	What You May Need to Know about Securing ADF Regions	16-12
16.2	Creating ADF Dynamic Regions	16-12
16.2.1	How to Create an ADF Dynamic Region.....	16-13
16.2.2	How to Override Parameter Values	16-14
16.3	Creating Dynamic Region Links	16-15

17 Creating Complex Task Flows

17.1	Using Initializers and Finalizers	17-1
17.2	Managing Transactions	17-2
17.2.1	How to Enable Transactions in an ADF Bounded Task Flow	17-3
17.2.2	What Happens When You Specify Transaction Options	17-4
17.3	Reentering an ADF Bounded Task Flow	17-5
17.3.1	How to Set Reentry Behavior	17-6
17.3.2	How to Set Outcome-Dependent Options	17-6
17.3.3	What You Should Know About Managed Bean Values Upon Task Flow Reentry	17-6
17.4	Handling Exceptions	17-7
17.4.1	Handling Exceptions During Transactions.....	17-8
17.4.2	What Happens When You Designate an Exception Handler	17-8
17.4.3	What You May Need to Know About Handling Validation Errors	17-8
17.5	Saving for Later	17-9
17.5.1	How to Add Save For Later Capabilities	17-11
17.5.2	How to Restore Save Points	17-13
17.5.3	How to Use the Save Point Restore Finalizer	17-13
17.5.4	How to Enable Implicit Save For Later.....	17-14
17.5.5	How to Set the Time-to-Live Period	17-14
17.5.6	What You Need to Know about Using Save for Later with BC4J.....	17-15
17.6	Creating a Train	17-15
17.6.1	ADF Bounded Task Flows as Trains	17-16
17.6.2	Train Sequences	17-18
17.6.3	How to Create a Train	17-19
17.6.4	What You May Need to Know About Grouping Activities	17-20
17.6.5	What You May Need to Know About Grouping Activities in Child Task Flows.	17-22
17.6.6	What You May Need To Know About Using Child Trains	17-23
17.6.7	What You May Need to Know About Branching	17-24
17.7	Running an ADF Bounded Task Flow as a Modal Dialog.....	17-24
17.7.1	How to Run an ADF Bounded Task Flow in a Modal Dialog	17-25
17.7.2	How to Pass Back a Return Value	17-26
17.8	Creating an ADF Task Flow Template	17-27
17.8.1	Copying and Referencing an ADF Task Flow Template	17-28

17.8.2	Creating an ADF Task Flow Template from Another Task Flow.....	17-30
17.8.3	How to Use an ADF Task Flow Template	17-30
17.8.4	How to Create an ADF Task Flow Template.....	17-30
17.8.5	What Happens When You Create an ADF Task Flow Template	17-31
17.8.6	What You Need to Know About ADF Task Flow Templates That Use Bindings	17-31
17.9	Creating a Page Hierarchy.....	17-32
17.9.1	XML Menu Model	17-33
17.9.2	How to Create a Page Hierarchy	17-34
17.10	How to Specify Bootstrap Configuration Files.....	17-38
17.11	Using the <code>adf-config.xml</code> File to Configure ADF Controller	17-39
17.11.1	Specifying Save for Later Settings	17-39
17.11.2	Validating ADF Controller Metadata	17-39
17.11.3	Searching for <code>adf-config.xml</code>	17-40
17.11.4	Replicating Memory Scopes in a Server-Cluster Environment	17-40

Part IV Creating a Databound Web User Interface

18 Getting Started with Your Web Interface

18.1	Introduction to Developing a Web Application with ADF Faces.....	18-1
18.2	Using Page Templates.....	18-1
18.2.1	How to Use ADF Data Binding in ADF Page Templates	18-3
18.2.2	What Happens When You Use ADF Model Layer Bindings on a Page Template .	18-4
18.2.3	What Happens at Runtime: How Pages Use Templates.....	18-5
18.3	Creating a Web Page	18-5
18.4	Using a Managed Bean in a Fusion Web Application.....	18-6
18.4.1	How to Use a Managed Bean to Store Information	18-8
18.4.2	What Happens When You Create a Managed Bean.....	18-9

19 Understanding the Fusion Page Lifecycle

19.1	Introduction to the Fusion Page Lifecycle.....	19-1
19.2	The JSF and ADF Page Lifecycles.....	19-3
19.2.1	What You May Need to Know About Using the Refresh Property Correctly.....	19-7
19.2.2	What You May Need to Know About Task Flows and the Lifecycle	19-8
19.3	Object Scope Lifecycles	19-9
19.3.1	What You May Need to Know About Object Scopes and Task Flows	19-11
19.4	Customizing the ADF Page Lifecycle	19-12
19.4.1	How to Create a Custom Phase Listener.....	19-12
19.4.2	How to Register a Listener Globally	19-12
19.4.3	What You May Need to Know About Listener Order	19-13
19.4.4	How To Register a Lifecycle Listener for a Single Page.....	19-14
19.4.5	What You May Need to Know About Extending RegionController for Page Fragments	
	19-15	

20 Creating a Basic Databound Page

20.1	Introduction to Creating a Basic Databound Page.....	20-1
20.2	Using Attributes to Create Text Fields.....	20-2

20.2.1	How to Create a Text Field	20-2
20.2.2	What Happens When You Create a Text Field	20-3
20.2.2.1	Creating and Using Iterator Bindings	20-3
20.2.2.2	Creating and Using Value Bindings	20-4
20.2.2.3	Using EL Expressions to Bind UI Components	20-5
20.3	Creating a Basic Form.....	20-6
20.3.1	How to Create a Form	20-6
20.3.2	What Happens When You Create a Form	20-7
20.4	Incorporating Range Navigation into Forms.....	20-9
20.4.1	How to Insert Navigation Controls into a Form	20-9
20.4.2	What Happens When You Create Command Buttons.....	20-10
20.4.2.1	Using Action Bindings for Built-in Navigation Operations.....	20-10
20.4.2.2	Iterator RangeSize Attribute	20-10
20.4.2.3	Using EL Expressions to Bind to Navigation Operations	20-11
20.4.2.4	Using Automatic Partial Page Rendering	20-13
20.4.3	What Happens at Runtime: How Action Events and Action Listeners Work	20-13
20.4.4	What You May Need to Know About the Browser Back Button and Navigating Through Records	20-14
20.5	Creating a Form to Edit an Existing Record	20-14
20.5.1	How to Create Edit Forms	20-15
20.5.2	What Happens When You Use Built-in Operations to Change Data	20-16
20.6	Creating an Input Form	20-18
20.6.1	How to Create an Input Form Using a Task Flow	20-18
20.6.2	What Happens When You Create an Input Form Using a Task Flow	20-19
20.6.3	What Happens At Runtime: CreateInsert Action from the Method Activity	20-20
20.6.4	What You May Need to Know About Displaying Sequence Numbers.....	20-20
20.7	Using a Dynamic Form to Determine Data to Display at Runtime.....	20-21
20.7.1	How to Use Dynamic Forms	20-21
20.7.2	What Happens When You Use Dynamic Components	20-22
20.7.3	What Happens at Runtime: How Attribute Values are Dynamically Determined	20-23
20.8	Modifying the UI Components and Bindings on a Form	20-23
20.8.1	How to Modify the UI Components and Bindings.....	20-23
20.8.2	What Happens When You Modify Attributes and Bindings	20-24

21 Creating ADF Databound Tables

21.1	Introduction to Adding Tables	21-1
21.2	Creating a Basic Table	21-2
21.2.1	How to Create a Basic Table.....	21-2
21.2.2	What Happens When You Create a Table	21-4
21.2.2.1	Iterator and Value Bindings for Tables	21-4
21.2.2.2	Code on the JSF Page for an ADF Faces Table	21-5
21.2.3	What You May Need to Know About Setting the Current Row in a Table	21-8
21.3	Creating an Editable Table	21-9
21.3.1	How to Create an Editable Table.....	21-11
21.3.2	What Happens When You Create an Editable Table	21-13
21.4	Creating an Input Table	21-13

21.4.1	How to Create an Input Table.....	21-13
21.4.2	What Happens When You Create an Input Table	21-14
21.4.3	What Happens at Runtime: How CreateInsert and Partial Page Refresh Work...	21-15
21.4.4	What You May Need to Know About Create and CreateInsert	21-15
21.5	Providing Multiselect Capabilities	21-16
21.5.1	How to Add Multiselect Capabilities	21-17
21.5.2	What Happens at Runtime: How an Operation Executes Against Multiple Rows	21-19
21.6	Modifying the Attributes Displayed in the Table	21-19
21.6.1	How to Modify the Displayed Attributes	21-19
21.6.2	How to Change the Binding for a Table	21-20
21.6.3	What Happens When You Modify Bindings or Displayed Attributes	21-20

22 Displaying Master-Detail Data

22.1	Introduction to Displaying Master-Detail Data.....	22-1
22.2	Identifying Master-Detail Objects on the Data Controls Panel.....	22-3
22.3	Using Tables and Forms to Display Master-Detail Objects	22-5
22.3.1	How to Display Master-Detail Objects in Tables and Forms	22-6
22.3.2	What Happens When You Create Master-Detail Tables and Forms	22-7
22.3.2.1	Code Generated in the JSF Page	22-7
22.3.2.2	Binding Objects Defined in the Page Definition File.....	22-8
22.3.3	What Happens at Runtime.....	22-9
22.3.4	What You May Need to Know About Master-Detail on Separate Pages	22-10
22.4	Using Trees to Display Master-Detail Objects.....	22-10
22.4.1	How to Display Master-Detail Objects in Trees.....	22-11
22.4.2	What Happens When You Create ADF Databound Trees	22-14
22.4.2.1	Code Generated in the JSF Page	22-14
22.4.2.2	Binding Objects Defined in the Page Definition File.....	22-15
22.4.3	What Happens at Runtime.....	22-16
22.5	Using Tree Tables to Display Master-Detail Objects	22-17
22.5.1	How to Display Master-Detail Objects in Tree Tables	22-18
22.5.2	What Happens When You Create a Databound Tree Table.....	22-18
22.5.2.1	Code Generated in the JSF Page	22-18
22.5.2.2	Binding Objects Defined in the Page Definition File.....	22-18
22.5.3	What Happens at Runtime.....	22-19
22.5.4	Using the TargetIterator Property	22-20

23 Creating Databound Selection Lists and Shuttles

23.1	Creating a Selection List.....	23-1
23.1.1	How to Create a Single Selection List	23-2
23.1.2	How to Create a Model-Driven List.....	23-4
23.1.3	How to Create a Selection List Containing Fixed Values	23-6
23.1.4	How to Create a Selection List Containing Dynamically Generated Values.....	23-7
23.1.5	What Happens When You Create a Model-driven Selection List	23-8
23.1.6	What Happens When You Create a Fixed Selection List.....	23-9
23.1.7	What You May Need to Know About Values in a Selection List	23-9

23.1.8	What Happens When You Create a Dynamic Selection List.....	23-9
23.2	Creating a List with Navigation List Binding.....	23-11
23.3	Creating a Databound Shuttle.....	23-11
23.3.1	How to Create a Shuttle.....	23-12

24 Creating Databound ADF Data Visualization Components

24.1	Introduction to Creating ADF Data Visualization Components	24-1
24.2	Creating Databound Graphs	24-3
24.2.1	How to Create a Graph	24-6
24.2.2	What Happens When You Use the Data Controls Panel to Create a Graph.....	24-8
24.2.3	What You May Need to Know About Using a Graph's Row Selection Listener for Master-Detail Processing 24-9	
24.3	Creating Databound Gauges.....	24-10
24.3.1	How to Create a Databound Dial Gauge	24-11
24.3.2	What Happens When You Create a Dial Gauge from a Data Control.....	24-13
24.3.3	How to Create a Databound Status Meter Gauge Set	24-14
24.3.4	What Happens When You Create a Status Meter Gauge from a Data Control....	24-16
24.4	Creating Databound Pivot Tables	24-17
24.4.1	How to Create a Pivot Table	24-18
24.4.2	What Happens When You Use the Data Controls Panel to Create a Pivot Table. 24-22	
24.4.2.1	Bindings for Pivot Tables	24-22
24.4.2.2	Code on the JSF Page for an ADF Pivot Table	24-23
24.4.3	What You May Need to Know About Aggregating Attributes in the Pivot Table 24-23	
24.4.3.1	Default Aggregation of Duplicate Data Rows.....	24-24
24.4.3.2	Custom Aggregation of Duplicate Rows	24-24
24.4.4	What You May Need to Know About Specifying an Initial Sort for a Pivot Table	
	24-25	
24.5	Creating Databound Geographic Maps.....	24-25
24.5.1	How to Create a Geographic Map with a Point Theme.....	24-27
24.5.2	How to Create Point Style Items for a Point Theme.....	24-29
24.5.3	What Happens When You Create a Geographic Map with a Point Theme.....	24-31
24.5.3.1	Binding XML for a Point Theme	24-31
24.5.3.2	XML Code on the JSF Page for a Geographic Map and Point Theme	24-31
24.5.4	What You May Need to Know About Adding Custom Point Style Items to a Map Point Theme 24-32	
24.5.5	How to Add a Databound Color Theme to a Geographic Map.....	24-33
24.5.6	What Happens When You Add a Color Theme to a Geographic Map	24-35
24.5.6.1	Binding XML for a Color Theme	24-35
24.5.6.2	XML Code on the JSF Page for a Color Theme	24-35
24.5.7	What You May Need to Know About Customizing Colors in a Map Color Theme	
	24-36	
24.5.8	How to Add a Databound Pie Graph Theme to a Geographic Map.....	24-36
24.5.9	What Happens When You Add a Pie Graph Theme to a Geographic Map	24-38
24.5.9.1	Binding XML for a Pie Graph Theme	24-38
24.5.9.2	Code on the JSF Page for a Pie Graph Theme	24-38
24.6	Creating Databound Gantt Charts	24-38
24.6.1	How to Create a Databound Project Gantt	24-39
24.6.2	What Happens When You Create a Project Gantt from a Data Control	24-42

24.6.3	What You May Need to Know About Summary Tasks in a Project Gantt	24-44
24.6.4	What You May Need to Know About Percent Complete in a Project Gantt	24-44
24.6.5	What You May Need to Know About Variance in a Project Gantt	24-45
24.6.6	How to Create a Databound Resource Utilization Gantt	24-45
24.6.7	What Happens When You Create a Resource Utilization Gantt	24-47
24.6.8	How to Create a Databound Scheduling Gantt.....	24-49
24.6.9	What Happens When You Create a Scheduling Gantt	24-52

25 Creating ADF Databound Search Forms

25.1	Introduction to Creating Search Forms	25-1
25.1.1	Query Search Forms	25-2
25.1.2	Quick Query Search Forms	25-7
25.1.3	Named Bind Variables in Query Search Forms	25-8
25.1.4	Filtered Table and Query-by-Example Searches.....	25-9
25.1.5	Implicit and Named View Criteria.....	25-11
25.1.6	List of Values (LOV) Input Fields.....	25-11
25.2	Creating Query Search Forms.....	25-13
25.2.1	How to Create a Query Search Form with a Results Table or Tree Table.....	25-14
25.2.2	How to Create a Query Search Form and Add a Results Component Later	25-14
25.2.3	How to Track Initial Query Execution.....	25-15
25.2.4	What Happens When You Create a Query Form	25-16
25.2.5	What Happens at Runtime: Creating a Search Form	25-16
25.3	Setting Up Search Form Properties	25-16
25.3.1	How to Set Search Form Properties on the View Criteria	25-17
25.3.2	How to Set Search Form Properties on the Query Component.....	25-18
25.3.3	How to Create Custom Operators or Remove Standard Operators	25-19
25.4	Creating Quick Query Search Forms	25-21
25.4.1	How to Create a Quick Query Search Form with a Results Table or Tree Table ..	25-21
25.4.2	How to Create a Quick Query Search Form and Add a Results Component Later	25-22
25.4.3	How to Set the Quick Query Layout Format.....	25-22
25.4.4	What Happens When You Create a Quick Query Search Form	25-22
25.4.5	What Happens at Runtime: Creating a Quick Query.....	25-23
25.5	Creating Filtered Search Tables	25-23

26 Creating More Complex Pages

26.1	Introduction to More Complex Pages.....	26-1
26.2	Creating Command Components to Execute Methods	26-2
26.2.1	How to Create a Command Component Bound to a Custom Method	26-2
26.2.2	What Happens When You Create Command Components Using a Method	26-3
26.2.2.1	Using Parameters in a Method	26-3
26.2.2.2	Using EL Expressions to Bind to Methods	26-3
26.2.2.3	Using the Return Value from a Method Call.....	26-4
26.2.3	What Happens at Runtime: Binding a Method to a Command Button.....	26-5
26.3	Setting Parameter Values Using a Command Component	26-5

26.3.1	How to Set Parameters Using setPropertyListener Within a Command Component	26-6
26.3.2	What Happens When You Set Parameters	26-6
26.3.3	What Happens at Runtime: Setting Parameters Using Command Component	26-6
26.4	Overriding Declarative Methods.....	26-6
26.4.1	How to Override a Declarative Method.....	26-7
26.4.2	What Happens When You Override a Declarative Method.....	26-9
26.5	Creating Contextual Events.....	26-10
26.5.1	How to Create Contextual Events.....	26-11
26.5.2	How to Manually Create the Event Map	26-12
26.5.3	What Happens When You Create Contextual Events.....	26-13
26.5.4	What Happens at Runtime: Contextual Events	26-14
26.6	Adding ADF Model Layer Validation.....	26-15
26.6.1	How to Add Validation	26-15
26.6.2	What Happens at Runtime: Model Validation Rules	26-16
26.7	Displaying Error Messages.....	26-16
26.8	Customizing Error Handling	26-17
26.8.1	Writing an Error Handler to Deal with Multiple Threads	26-18

27 Designing a Page Using Placeholder Data Controls

27.1	Introduction to Placeholder Data Controls	27-1
27.2	Creating Placeholder Data Controls.....	27-2
27.2.1	How to Create a Placeholder Data Control.....	27-2
27.2.2	What Happens When You Create a Placeholder Data Control	27-3
27.3	Creating Placeholder Data Types	27-4
27.3.1	How to Create a Placeholder Data Type	27-5
27.3.2	What Happens When You Create a Placeholder Data Type	27-7
27.3.3	How to Configure a Placeholder Data Type Attribute to Be an LOV	27-9
27.3.4	How to Create Master-Detail Data Types	27-11
27.3.5	What Happens When You Create a Master-Detail Data Type.....	27-13
27.3.6	How to Add Sample Data	27-14
27.3.7	What Happens When You Add Sample Data	27-15
27.4	Using Placeholder Data Controls	27-16
27.4.1	Limitations of Placeholder Data Controls.....	27-16
27.4.2	Creating Layout	27-16
27.4.3	Creating a Search Form.....	27-17
27.4.4	Binding Components	27-17
27.4.5	Rebinding Components	27-17
27.4.6	Packaging Placeholder Data Controls to ADF Library JARs	27-17

Part V Completing Your Application

28 Adding Security to a Fusion Web Application

28.1	Introduction to ADF Security for Fusion Web Applications.....	28-1
28.2	Choosing ADF Security Authentication and Authorization	28-2
28.2.1	How to Enable Only ADF Authentication	28-6
28.2.2	What Happens When You Choose Not to Enforce Authorization.....	28-8

28.2.3	What You May Need to Know About the valid-users Role	28-10
28.2.4	How to Enable ADF Authentication and Authorization	28-11
28.2.5	What Happens When You Choose to Enforce Authorization	28-13
28.2.6	How to Disable ADF Security	28-15
28.2.7	What Happens When You Disable ADF Security	28-16
28.2.8	What Happens at Runtime: How Oracle ADF Security Handles Authentication	28-16
28.2.9	What Happens at Runtime: How Oracle ADF Security Handles Authorization..	28-18
28.3	Defining Users and Roles for a Fusion Web Application	28-20
28.3.1	How to Enable the test-all Application Role in JDeveloper	28-20
28.3.2	How to Define Application Roles in JDeveloper	28-22
28.3.3	How to Configure the Identity Store with Test Users in JDeveloper.....	28-26
28.4	Defining ADF Security Access Policies.....	28-31
28.4.1	How to Grant Permissions on ADF Bounded Task Flows	28-33
28.4.2	How to Grant Permissions on Individual Web Pages Using ADF Page Definitions	28-34
28.4.3	How to Secure Row Data Using ADF Business Components.....	28-37
28.4.3.1	Defining a Permission on ADF Business Component Entity Objects.....	28-37
28.4.3.2	Granting Permissions on ADF Business Components.....	28-39
28.4.4	What Happens When You Define a Security Policy for ADF Resources	28-40
28.4.5	What You May Need to Know About ADF Resource Grants.....	28-41
28.4.6	How to Use Regular Expressions to Define Policies on Groups of Resources	28-42
28.5	Handling User Authentication in a Fusion Web Application.....	28-44
28.5.1	How to Create a Login Component for Your Application	28-44
28.5.2	How to Add a Login Component to a Web Page	28-49
28.5.3	How to Create a Login Page for Your Application.....	28-51
28.5.3.1	Creating an Oracle ADF Faces--Based Login Page.....	28-52
28.5.3.2	Creating Login Code for the Backing Bean	28-53
28.5.3.3	Configuring the web.xml File for an Oracle ADF Faces-Based Login Page ...	28-55
28.5.3.4	Ensuring That the Login Page Is Public	28-56
28.5.4	How to Create a Public Welcome Page for Your Application	28-56
28.5.4.1	Ensuring That the Welcome Page Is Public	28-56
28.5.4.2	Adding Login and Logout Links.....	28-56
28.5.4.3	Hiding Links to Secured Pages.....	28-57
28.5.5	What You May Need to Know About the Anonymous User	28-57
28.6	Performing Authorization Checks in a Fusion Web Application.....	28-57
28.6.1	How to Evaluate Policies Using Expression Language (EL)	28-58
28.6.2	What You May Need to Know About Delayed Evaluation of EL.....	28-60
28.6.3	How to Evaluate Policies Using Java.....	28-61
28.7	Getting Other Information from the Oracle ADF Security Context	28-62
28.7.1	How to Determine Whether Security Is Enabled	28-62
28.7.2	How to Determine Whether the User Is Authenticated.....	28-62
28.7.3	How to Determine the Current User Name.....	28-62
28.7.4	How to Determine Membership of a Java EE Security Role	28-63
28.8	Testing ADF Security with Integrated WLS in JDeveloper	28-63
28.9	Preparing for Deployment to a Production Environment	28-65

29 Testing and Debugging ADF Components

29.1	Introduction to Oracle ADF Debugging	29-1
29.2	Correcting Simple Oracle ADF Compilation Errors	29-2
29.3	Correcting Simple Oracle ADF Runtime Errors	29-4
29.4	Using the ADF Logger and Business Component Browser.....	29-6
29.4.1	How to Turn On Diagnostic Logging	29-6
29.4.2	How to Create an Oracle ADF Debugging Configuration	29-6
29.4.3	How to Use the Business Component Browser with the Debugger	29-8
29.5	Using the ADF Declarative Debugger	29-9
29.5.1	Using ADF Source Code with the Debugger.....	29-10
29.5.2	How to Set Up the ADF Source User Library.....	29-11
29.5.3	How to Add the ADF Source Library to a Project	29-11
29.5.4	How to Use the EL Expression Evaluator	29-12
29.5.5	How to View and Export Stack Trace Information	29-13
29.6	Setting ADF Declarative Breakpoints	29-13
29.6.1	How to Set Task Flow Activity Breakpoints.....	29-19
29.6.2	How to Set Page Definition Executable Breakpoints.....	29-20
29.6.3	How to Set Page Definition Action Binding Breakpoints.....	29-21
29.6.4	How to Set Page Definition Attribute Value Binding Breakpoints.....	29-22
29.6.5	How to Use the ADF Structure Window	29-23
29.6.6	How to Use the ADF Data Window	29-25
29.7	Setting Java Code Breakpoints.....	29-29
29.7.1	How to Set Java Breakpoints on Classes and Methods.....	29-29
29.7.2	How to Optimize Use of the Source Editor	29-30
29.7.3	How to Set Breakpoints and Debug Using ADF Source Code	29-31
29.7.4	How to Use Debug Libraries for Symbolic Debugging	29-31
29.7.5	How to Use Different Kinds of Breakpoints.....	29-33
29.7.6	How to Edit Breakpoints for Improved Control	29-34
29.7.7	How to Filter Your View of Class Members.....	29-35
29.7.8	How to Use Common Oracle ADF Breakpoints	29-35
29.8	Regression Testing with JUnit.....	29-36
29.8.1	How to Obtain the JUnit Extension.....	29-37
29.8.2	How to Create a JUnit Test Case	29-38
29.8.3	How to Create a JUnit Test Fixture	29-40
29.8.4	How to Create a JUnit Test Suite.....	29-40
29.8.5	How to Run a JUnit Test Suite as Part of an Ant Build Script	29-41

30 Refactoring a Fusion Web Application

30.1	Introduction to Refactoring a Fusion Web Application	30-1
30.2	Renaming Files	30-1
30.3	Moving JSF Pages	30-2
30.4	Refactoring pagedef.xml Bindings Objects	30-2
30.5	Refactoring ADF Business Components	30-3
30.6	Refactoring Attributes	30-3
30.7	Refactoring Named Elements	30-4
30.8	Refactoring ADF Task Flows.....	30-5
30.9	Refactoring the DataBindings.cpx File.....	30-5

30.10	Refactoring Across Abstraction Layers	30-6
30.11	Refactoring Limitations	30-6
30.12	Refactoring the .jpx Project File	30-7

31 Reusing Application Components

31.1	Introduction to Reusable Components.....	31-1
31.1.1	Creating Reusable Components	31-3
31.1.1.1	Naming Conventions.....	31-3
31.1.1.2	The Naming Process for the ADF Library JAR Deployment Profile	31-5
31.1.1.3	Keeping the Relevant Project.....	31-6
31.1.1.4	Selecting the Relevant Technology Scope.....	31-6
31.1.1.5	Selecting Paths and Folders	31-6
31.1.1.6	Including Connections Within Reusable Components.....	31-6
31.1.2	Using the Resource Palette	31-7
31.1.3	Extension Libraries	31-8
31.2	Packaging a Reusable ADF Component into an ADF Library.....	31-12
31.2.1	How to Package a Component into an ADF Library JAR	31-12
31.2.2	What Happens When You Package a Project to an ADF Library JAR	31-16
31.2.2.1	Application Modules	31-17
31.2.2.2	Data Controls	31-17
31.2.2.3	Task Flows	31-17
31.2.2.4	Page Templates	31-17
31.2.2.5	Declarative Components	31-18
31.3	Adding ADF Library Components into Projects.....	31-18
31.3.1	How to Add an ADF Library JAR into a Project using the Resource Palette	31-18
31.3.2	How to Add an ADF Library JAR into a Project Manually.....	31-19
31.3.3	What Happens When You Add an ADF Library JAR to a Project	31-20
31.3.4	What You May Need to Know About Using ADF Library Components	31-22
31.3.4.1	Using Data Controls.....	31-23
31.3.4.2	Using Application Modules.....	31-23
31.3.4.3	Using Business Components	31-23
31.3.4.4	Using Task Flows.....	31-24
31.3.4.5	Using Page Templates.....	31-24
31.3.4.6	Using Declarative Components.....	31-25
31.3.5	What You May Need to Know About Differentiating ADF Library Components	31-25
31.3.6	What Happens at Runtime: Adding ADF Libraries	31-25
31.4	Removing an ADF Library JAR from a Project	31-26
31.4.1	How to Remove an ADF Library JAR from a Project Using the Resource Palette	31-26
31.4.2	How to Remove an ADF Library JAR from a Project Manually.....	31-26

32 Deploying Fusion Web Applications

32.1	Introduction to Deploying Fusion Web Applications	32-1
32.2	Creating a Connection to the Target Application Server	32-3
32.3	Creating a Deployment Profile	32-4
32.3.1	How to Create Deployment Profiles.....	32-4

32.3.2	How to View and Change Deployment Profile Properties	32-5
32.4	Creating and Editing Deployment Descriptors.....	32-6
32.4.1	How to Create Deployment Descriptors.....	32-6
32.4.2	How to View or Change Deployment Descriptor Properties	32-7
32.4.3	How to Create or Configure the application.xml file for WebLogic Compatibility	32-7
32.4.4	How to Create or Configure the web.xml file for WebLogic Compatibility.....	32-8
32.4.5	How to Create Deployment Descriptors for Applications with ADF Faces Components 32-8	
32.5	Installing the ADF Runtime to the WebLogic Installation.....	32-8
32.5.1	How to Install the ADF Runtime into an Existing WebLogic Server.....	32-8
32.5.2	What You May Need to Know About ADF Libraries	32-9
32.6	Creating and Extending WebLogic Domains	32-9
32.6.1	How to Create a WebLogic Domain for Oracle ADF	32-9
32.6.2	How to Extend a WebLogic Domain for Oracle ADF	32-10
32.7	Creating a JDBC Data Source for a WebLogic Domain Server	32-11
32.8	Deploying the Application	32-12
32.8.1	How to Deploy to a WebLogic Server from JDeveloper	32-13
32.8.2	How to Create an EAR File for Deployment	32-14
32.8.3	How to Deploy an Application Using Ant	32-14
32.9	Testing the Application and Verifying Deployment	32-14

Part VI Advanced Topics

33 Advanced Business Components Techniques

33.1	Globally Extending ADF Business Components Functionality	33-1
33.1.1	What Are ADF Business Components Framework Extension Classes?.....	33-1
33.1.2	How To Create a Framework Extension Class.....	33-2
33.1.3	What Happens When You Create a Framework Extension Class.....	33-3
33.1.4	How to Base an ADF Component on a Framework Extension Class	33-3
33.1.5	What Happens When You Base a Component on a Framework Extension Class..	33-4
33.1.5.1	Basing an XML-Only Component on a Framework Extension Class.....	33-4
33.1.5.2	Basing a Component with a Custom Java Class on a Framework Extension Class ... 33-5	
33.1.6	What You May Need to Know.....	33-6
33.1.6.1	Don't Update the Extends Clause in Custom Component Java Files By Hand	33-6
33.1.6.2	You Can Have Multiple Levels of Framework Extension Classes.....	33-6
33.1.6.3	Setting up Project-Level Preferences for Framework Extension Classes	33-7
33.1.6.4	Setting Up Framework Extension Class Preferences at the IDE Level.....	33-7
33.2	Creating a Layer of Framework Extensions.....	33-7
33.2.1	How to Create Your Layer of Framework Extension Layer Classes.....	33-8
33.2.2	How to Package Your Framework Extension Layer in a JAR File	33-9
33.2.3	How to Create a Library Definition for Your Framework Extension JAR File.....	33-9
33.3	Customizing Framework Behavior with Extension Classes.....	33-10
33.3.1	How to Access Runtime Metadata For View Objects and Entity Objects	33-10
33.3.2	Implementing Generic Functionality Using Runtime Metadata	33-11
33.3.3	Implementing Generic Functionality Driven by Custom Properties.....	33-12
33.3.4	What You May Need to Know.....	33-13

33.3.4.1	Determining the Attribute Kind at Runtime	33-13
33.3.4.2	Configuring Design Time Custom Property Names.....	33-13
33.3.4.3	Setting Custom Properties at Runtime	33-13
33.4	Creating Generic Extension Interfaces.....	33-13
33.5	Invoking Stored Procedures and Functions.....	33-16
33.5.1	Invoking Stored Procedures with No Arguments	33-16
33.5.2	Invoking Stored Procedure with Only IN Arguments.....	33-16
33.5.3	Invoking Stored Function with Only IN Arguments	33-17
33.5.4	Calling Other Types of Stored Procedures	33-19
33.6	Accessing the Current Database Transaction	33-20
33.7	Working with Libraries of Reusable Business Components	33-21
33.7.1	How To Create a Reusable Library of Business Components	33-21
33.7.2	How To Import a Package of Reusable Components from a Library.....	33-23
33.7.3	How to Remove an Imported Package from a Project	33-23
33.7.4	What Happens When You Import a Package of Reusable Components from a Library..	33-24
33.7.5	What You May Need to Know.....	33-24
33.7.5.1	Components in Imported Libraries Are Not Editable	33-24
33.7.5.2	Have to Close/Reopen to See Changes from a JAR.....	33-24
33.8	Customizing Business Components Error Messages	33-24
33.8.1	How to Customize Base ADF Business Components Error Messages	33-25
33.8.2	What Happens When You Customize Base ADF Business Components Error Messages	33-26
33.8.3	How to Customize Error Messages for Database Constraint Violations	33-26
33.8.4	How to Implement a Custom Constraint Error Handling Routine	33-27
33.8.4.1	Creating a Custom Database Transaction Framework Extension Class	33-27
33.8.4.2	Configuring an Application Module to Use a Custom Database Transaction Class .	33-28
33.9	Creating Extended Components Using Inheritance	33-29
33.9.1	How To Create a Component That Extends Another	33-29
33.9.2	How To Extend a Component After Creation.....	33-30
33.9.3	What Happens When You Create a Component That Extends Another	33-30
33.9.3.1	Understanding an Extended Component's XML Descriptor	33-31
33.9.3.2	Understanding Java Code Generation for an Extended Component	33-31
33.9.4	What You May Need to Know.....	33-31
33.9.4.1	You Can Use Parent Classes and Interfaces to Work with Extended Components ...	33-32
33.9.4.2	Class Extends is Disabled for Extended Components	33-34
33.9.4.3	Interesting Aspects You Can Extend for Key Component Types	33-34
33.9.4.4	Extended Components Have Attribute Indices Relative to Parent.....	33-34
33.10	Substituting Extended Components In a Delivered Application	33-35
33.10.1	Extending and Substituting Components Is Superior to Modifying Code.....	33-35
33.10.2	How To Substitute an Extended Component.....	33-35
33.10.3	What Happens When You Substitute.....	33-36
33.10.4	Enabling the Substituted Components in the Base Application.....	33-36

34 Advanced Entity Object Techniques

34.1	Creating Custom, Validated Data Types Using Domains	34-1
34.1.1	What Are Domains?	34-2
34.1.2	How To Create a Domain	34-2
34.1.3	What Happens When You Create a Domain	34-2
34.1.4	What You May Need to Know About Domains	34-3
34.1.4.1	Using Domains for Entity and View Object Attributes	34-3
34.1.4.2	Validate Method Should Throw DataCreationException If Sanity Checks Fail	34-3
34.1.4.3	String Domains Aggregate a String Value	34-4
34.1.4.4	Other Domains Extend Existing Domain Type	34-4
34.1.4.5	Simple Domains are Immutable Java Classes	34-5
34.1.4.6	Creating Domains for Oracle Object Types When Useful	34-5
34.1.4.7	Quickly Navigating to the Domain Class	34-6
34.1.4.8	Domains Get Packaged in the Common JAR	34-6
34.1.4.9	Entity and View Object Attributes Inherit Custom Domain Properties	34-6
34.1.4.10	Domain Settings Cannot Be Less Restrictive at Entity or View Level	34-6
34.2	Updating a Deleted Flag Instead of Deleting Rows	34-7
34.2.1	How to Update a Deleted Flag When a Row is Removed	34-7
34.2.2	Forcing an Update DML Operation Instead of a Delete	34-7
34.3	Using Update Batching	34-8
34.4	Advanced Entity Association Techniques	34-8
34.4.1	Modifying Association SQL Clause to Implement Complex Associations	34-8
34.4.2	Exposing View Link Accessor Attributes at the Entity Level	34-9
34.4.3	Optimizing Entity Accessor Access By Retaining the Row Set	34-9
34.5	Basing an Entity Object on a PL/SQL Package API	34-10
34.5.1	How to Create an Entity Object Based on a View	34-11
34.5.2	What Happens When You Create an Entity Object Based on a View	34-11
34.5.3	Centralizing Details for PL/SQL-Based Entities into a Base Class	34-11
34.5.4	Implementing the Stored Procedure Calls for DML Operations	34-12
34.5.5	Adding Select and Lock Handling	34-13
34.5.5.1	Updating PLSQLEntityImpl Base Class to Handle Lock and Select	34-13
34.5.5.2	Implementing Lock and Select for the Product Entity	34-14
34.5.5.3	Refreshing the Entity Object After RowInconsistentException	34-17
34.6	Basing an Entity Object on a Join View or Remote DBLink	34-17
34.7	Using Inheritance in Your Business Domain Layer	34-18
34.7.1	Understanding When Inheritance Can be Useful	34-18
34.7.2	How To Create Entity Objects in an Inheritance Hierarchy	34-19
34.7.2.1	Start By Identifying the Discriminator Column and Distinct Values	34-19
34.7.2.2	Identify the Subset of Attributes Relevant to Each Kind of Entity	34-20
34.7.2.3	Creating the Base Entity Object in an Inheritance Hierarchy	34-20
34.7.2.4	Creating a Subtype Entity Object in an Inheritance Hierarchy	34-21
34.7.3	How to Add Methods to Entity Objects in an Inheritance Hierarchy	34-22
34.7.3.1	Adding Methods Common to All Entity Objects in the Hierarchy	34-22
34.7.3.2	Overriding Common Methods in a Subtype Entity	34-22
34.7.3.3	Adding Methods Specific to a Subtype Entity	34-23
34.7.4	What You May Need to Know About Using Inheritance	34-23

34.7.4.1	Sometimes You Need to Introduce a New Base Entity.....	34-23
34.7.4.2	Finding Subtype Entities by Primary Key	34-23
34.7.4.3	You Can Create View Objects with Polymorphic Entity Usages	34-24
34.8	Controlling Entity Posting Order to Avoid Constraint Violations.....	34-24
34.8.1	Understanding the Default Post Processing Order	34-24
34.8.2	How Compositions Change the Default Processing Ordering	34-24
34.8.3	Overriding postChanges() to Control Post Order.....	34-24
34.8.3.1	Observing the Post Ordering Problem First Hand	34-24
34.8.3.2	Forcing the Supplier to Post Before the Product.....	34-26
34.8.3.3	Understanding Associations Based on DBSequence-Valued Primary Keys ..	34-27
34.8.3.4	Refreshing References to DBSequence-Assigned Foreign Keys.....	34-28
34.9	Implementing Automatic Attribute Recalculation	34-29
34.10	Implementing Custom Validation Rules.....	34-31
34.10.1	How To Create a Custom Validation Rule	34-31
34.10.2	Adding a Design Time Bean Customizer for Your Rule.....	34-33
34.10.3	Registering and Using a Custom Rule in JDeveloper	34-34
34.11	Creating New History Types	34-35
34.11.1	How to Create New History Types.....	34-35
34.11.2	How to Remove a History Type	34-36

35 Advanced View Object Techniques

35.1	Advanced View Object Concepts and Features	35-1
35.1.1	Using a Max Fetch Size to Only Fetch the First N Rows.....	35-1
35.1.2	Consistently Displaying New Rows in View Objects Based on the Same Entity....	35-2
35.1.2.1	How View Link Consistency Mode Works	35-2
35.1.2.2	Understanding the Default View Link Consistency Setting and How to Change It..	35-2
35.1.2.3	Using a RowMatch to Qualify Which New, Unposted Rows Get Added to a Row Set	35-3
35.1.2.4	Setting a Dynamic Where Clause Disables View Link Consistency	35-4
35.1.2.5	New Row from Other View Objects Added at the Bottom.....	35-4
35.1.2.6	New, Unposted Rows Added to Top of RowSet when Re-Executed	35-4
35.1.3	Understanding View Link Accessors Versus Data Model View Link Instances....	35-5
35.1.3.1	Enabling a Dynamic Detail Row Set with Active Master/Detail Coordination	35-5
35.1.3.2	Accessing a Stable Detail Row Set Using View Link Accessor Attributes.....	35-5
35.1.3.3	Accessor Attributes Create Distinct Row Sets Based on an Internal View Object.....	35-6
35.1.4	Presenting and Scrolling Data a Page at a Time Using the Range	35-7
35.1.5	Efficiently Scrolling Through Large Result Sets Using Range Paging.....	35-8
35.1.5.1	Understanding How to Oracle Supports "TOP-N" Queries.....	35-9
35.1.5.2	How to Enable Range Paging for a View Object	35-10
35.1.5.3	What Happens When You Enable Range Paging	35-11
35.1.5.4	What Happens When View Rows are Cached When Using Range Paging ...	35-12
35.1.5.5	How to Scroll to a Given Page Number Using Range Paging	35-12
35.1.5.6	Estimating the Number of Pages in the Row Set Using Range Paging	35-12
35.1.5.7	Understanding the Tradeoffs of Using a Range Paging Mode	35-12

35.1.6	Setting Up a Data Model with Multiple Masters	35-13
35.1.7	Understanding When You Can Use Partial Keys with <code>findByPrimaryKey()</code>	35-14
35.1.8	Creating Dynamic Attributes to Store UI State	35-15
35.1.9	Working with Multiple Row Sets and Row Set Iterators.....	35-15
35.1.10	Optimizing View Link Accessor Access By Retaining the Row Set	35-15
35.2	Tuning Your View Objects for Best Performance	35-17
35.2.1	Use Bind Variables for Parameterized Queries.....	35-17
35.2.1.1	Use Bind Variables to Avoid Re-parsing of Queries	35-17
35.2.1.2	Use Bind Variables to Prevent SQL-Injection Attacks	35-17
35.2.2	Consider Using Entity-Based View Objects for Read-Only Data	35-18
35.2.3	Use SQL Tracing to Identify Ill-Performing Queries.....	35-20
35.2.4	Consider the Appropriate Tuning Settings for Every View Object	35-21
35.2.4.1	Set the Database Retrieval Options Appropriately	35-21
35.2.4.2	Consider Whether Fetching One Row at a Time is Appropriate	35-22
35.2.4.3	Specify a Query Optimizer Hint if Necessary	35-22
35.2.5	Creating View Objects at Design Time	35-23
35.2.6	Use Forward Only Mode to Avoid Caching View Rows.....	35-23
35.3	Using Expert Mode for Full Control Over SQL Query	35-23
35.3.1	How to Enable Expert Mode for Full SQL Control.....	35-24
35.3.2	What Happens When You Enable Expert Mode.....	35-24
35.3.3	What You May Need to Know.....	35-25
35.3.3.1	You May Need to Perform Manual Attribute Mapping.....	35-25
35.3.3.2	Disabling Expert Mode Loses Any Custom Edits	35-26
35.3.3.3	Once In Expert Mode, Changes to SQL Expressions Are Ignored	35-26
35.3.3.4	Don't Map Incorrect Calculated Expressions to Entity Attributes.....	35-27
35.3.3.5	Expert Mode SQL Formatting is Retained.....	35-28
35.3.3.6	Expert Mode Queries Are Wrapped as Inline Views	35-28
35.3.3.7	Disabling the Use of Inline View Wrapping at Runtime	35-29
35.3.3.8	Enabling Expert Mode May Impact Dependent Objects	35-29
35.4	Generating Custom Java Classes for a View Object	35-30
35.4.1	How To Generate Custom Classes.....	35-30
35.4.1.1	Generating Bind Variable Accessors.....	35-30
35.4.1.2	Generating View Row Attribute Accessors	35-31
35.4.1.3	Exposing View Row Accessors to Clients.....	35-32
35.4.1.4	Configuring Default Java Generation Preferences	35-33
35.4.2	What Happens When You Generate Custom Classes.....	35-33
35.4.2.1	Seeing and Navigating to Custom Java Files	35-33
35.4.3	What You May Need to Know About Custom Classes	35-34
35.4.3.1	About the Framework Base Classes for a View Object	35-34
35.4.3.2	You Can Safely Add Code to the Custom Component File	35-34
35.4.3.3	Attribute Indexes and <code>InvokeAccessor</code> Generated Code	35-34
35.5	Working Programmatically with Multiple Named View Criteria	35-36
35.5.1	Applying One or More Named View Criteria.....	35-36
35.5.2	Removing All Applied Named View Criteria	35-37
35.5.3	Using the Named Criteria at Runtime	35-37
35.6	Performing In-Memory Sorting and Filtering of Row Sets.....	35-38
35.6.1	Understanding the View Object's SQL Mode.....	35-38

35.6.2	Sorting View Object Rows In Memory	35-39
35.6.2.1	Combining setSortBy and setQueryMode for In-Memory Sorting.....	35-39
35.6.2.2	Extensibility Points for In-Memory Sorting.....	35-40
35.6.3	Performing In-Memory Filtering with View Criteria.....	35-41
35.6.4	Performing In-Memory Filtering with RowMatch	35-43
35.6.4.1	Applying a RowMatch to a View Object.....	35-44
35.6.4.2	Using RowMatch to Test an Individual Row	35-45
35.6.4.3	How a RowMatch Affects Rows Fetched from the Database	35-45
35.7	Using View Objects to Work with Multiple Row Types.....	35-46
35.7.1	Working with Polymorphic Entity Usages	35-46
35.7.2	How To Create a View Object with a Polymorphic Entity Usage.....	35-46
35.7.3	What Happens When You Create a View Object with a Polymorphic Entity Usage.....	35-47
35.7.4	What You May Need to Know.....	35-47
35.7.4.1	Your Query Must Limit Rows to Expected Entity Subtypes	35-47
35.7.4.2	Exposing Selected Entity Methods in View Rows Using Delegation	35-47
35.7.4.3	Creating New Rows With the Desired Entity Subtype.....	35-49
35.7.5	Working with Polymorphic View Rows	35-49
35.7.6	How to Create a View Object with Polymorphic View Rows.....	35-50
35.7.7	What You May Need to Know.....	35-51
35.7.7.1	Selecting Subtype-Specific Attributes in Extended View Objects	35-51
35.7.7.2	Delegating to Subtype-Specific Methods After Overriding the Entity Usage	35-52
35.7.7.3	Working with Different View Row Interface Types in Client Code	35-52
35.7.7.4	View Row Polymorphism and Polymorphic Entity Usage are Orthogonal...	35-54
35.8	Reading and Writing XML	35-54
35.8.1	How to Produce XML for Queried Data	35-54
35.8.2	What Happens When You Produce XML	35-55
35.8.3	What You May Need to Know.....	35-57
35.8.3.1	Controlling XML Element Names.....	35-57
35.8.3.2	Controlling Element Suppression for Null-Valued Attributes.....	35-58
35.8.3.3	Printing or Searching the Generated XML Using XPath	35-58
35.8.3.4	Using the Attribute Map For Fine Control Over Generated XML	35-59
35.8.3.5	Use the Attribute Map Approach with Bi-Directional View Links.....	35-59
35.8.3.6	Transforming Generated XML Using an XSLT Stylesheet	35-60
35.8.3.7	Generating XML for a Single Row	35-61
35.8.4	How to Consume XML Documents to Apply Changes.....	35-61
35.8.5	What Happens When You Consume XML Documents.....	35-61
35.8.5.1	How ViewObject.readXML() Processes an XML Document	35-61
35.8.5.2	Using readXML() to Processes XML for a Single Row.....	35-62
35.9	Using Programmatic View Objects for Alternative Data Sources	35-64
35.9.1	How to Create a Read-Only Programmatic View Object	35-64
35.9.2	How to Create an Entity-Based Programmatic View Object.....	35-65
35.9.3	Key Framework Methods to Override for Programmatic View Objects.....	35-65
35.9.4	How to Create a View Object on a REF CURSOR	35-66
35.9.4.1	The Overridden create() Method	35-67
35.9.4.2	The Overridden executeQueryForCollection() Method	35-67
35.9.4.3	The Overridden createRowFromResultSet() Method	35-67

35.9.4.4	The Overridden hasNextForCollectionMethod()	35-68
35.9.4.5	The Overridden releaseUserDataForCollection() Method	35-68
35.9.4.6	The Overridden getQueryHitCount() Method	35-69
35.10	Creating a View Object with Multiple Updatable Entities	35-69
35.11	Declaratively Preventing Insert, Update, and Delete	35-71

36 Application State Management

36.1	Understanding Why State Management is Necessary	36-1
36.1.1	Examples of Multi-Step Tasks.....	36-1
36.1.2	Stateless HTTP Protocol Complicates Stateful Applications.....	36-2
36.1.3	How Cookies Are Used to Track a User Session.....	36-2
36.1.4	Performance and Reliability Impact of Using HttpSession	36-3
36.2	About Fusion Web Application State Management	36-5
36.2.1	Basic Architecture of the Save for Later Facility	36-5
36.2.2	Basic Architecture of the Application Module State Management Facility	36-5
36.2.2.1	Understanding When Passivation and Activation Occurs.....	36-6
36.2.2.2	How Passivation Changes When Optional Failover Mode is Enabled	36-8
36.2.2.3	About State Management Release Levels	36-8
36.3	Using Save For Later	36-10
36.4	Setting the Application Module Release Level at Runtime	36-11
36.4.1	How to Set Unmanaged Level	36-11
36.4.2	How to Set Reserved Level.....	36-11
36.4.3	How to Set Managed Level	36-11
36.4.4	How to Set Release Level in a JSF Backing Bean.....	36-11
36.4.5	How to Set Release Level in an ADF PagePhaseListener	36-12
36.4.6	How to Set Release Level in an ADF PageController.....	36-12
36.4.7	How to Set Release Level in an Custom ADF PageLifecycle	36-13
36.5	What Model State Is Saved and When It Is Cleaned Up	36-14
36.5.1	What State is Saved?.....	36-14
36.5.2	Where is the Model State Saved?.....	36-15
36.5.2.1	How Database-Backed Passivation Works	36-15
36.5.2.2	Controlling the Schema Where the State Management Table Resides	36-15
36.5.2.3	Configuring the Type of Passivation Store	36-15
36.5.3	When is the Model State Cleaned Up?	36-16
36.5.3.1	Previous Snapshot Removed When Next One Taken.....	36-16
36.5.3.2	Passivation Snapshot Removed on Unmanaged Release	36-16
36.5.3.3	Passivation Snapshot Retained in Failover Mode	36-17
36.5.4	Cleaning Up Temporary Storage Tables	36-17
36.6	Timing Out the HttpSession.....	36-18
36.6.1	How to Configure the Implicit Timeout Due to User Inactivity.....	36-18
36.6.2	How to Code an Explicit HttpSession Timeout	36-18
36.7	Managing Custom User-Specific Information	36-19
36.7.1	How to Passivate Custom User-Specific Information	36-19
36.7.1.1	What Happens When You Passivate Custom Information.....	36-20
36.8	Managing the State of View Objects	36-21
36.8.1	How to Manage the State of View Objects.....	36-21
36.8.2	What You May Need to Know About Passivating View Objects	36-21

36.8.3	How to Manage the State of Transient View Objects and Attributes	36-23
36.8.4	What You May Need to Know About Passivating Transient View Objects	36-23
36.8.5	How to Use Transient View Objects to Store Session-level Global Variables	36-23
36.9	Using State Management for Middle-Tier Savepoints	36-25
36.9.1	How to Use State Management for Savepoints.....	36-25
36.10	Testing to Ensure Your Application Module is Activation-Safe.....	36-25
36.10.1	Understanding the jbo.ampool.doampooling Configuration Parameter	36-25
36.10.2	Disabling Application Module Pooling to Test Activation	36-25
36.11	Keeping Pending Changes in the Middle Tier	36-26
36.11.1	How to Set Applications to Use Optimistic Locking.....	36-27
36.11.2	How to Avoid Clashes Using the postChanges() Method.....	36-27
36.11.3	How to Use the Reserved Level For Pending Database States	36-28

37 Understanding Application Module Pooling

37.1	Overview of Application Module Pooling.....	37-1
37.2	Understanding Configuration Property Scopes.....	37-2
37.3	Setting Pool Configuration Parameters	37-3
37.3.1	Setting Configuration Properties Declaratively	37-3
37.3.2	Setting Configuration Properties as System Parameters	37-5
37.3.3	Programmatically Setting Configuration Properties.....	37-6
37.4	How Many Pools are Created, and When?	37-7
37.4.1	Application Module Pools.....	37-7
37.4.2	Database Connection Pools	37-7
37.4.3	Understanding Application Module and Connection Pools.....	37-8
37.4.3.1	Single Oracle Application Server Instance, Single Oracle WebLogic Server Instance, Single JVM	37-8
37.4.3.2	Multiple Oracle Application Server Instances, Single Oracle WebLogic Server Instance, Multiple JVMs	37-9
37.5	Application Module Pool Parameters.....	37-10
37.5.1	Pool Behavior Parameters	37-10
37.5.2	Pool Sizing Parameters	37-11
37.5.3	Pool Cleanup Parameters	37-12
37.6	Database Connection Pool Parameters	37-14
37.7	How Database and Application Module Pools Cooperate.....	37-16
37.8	Database User State and Pooling Considerations	37-17
37.8.1	How Often prepareSession() Fires When jbo.doconnectionpooling = false	37-18
37.8.2	How to Set Database User State When jbo.doconnectionpooling = true	37-18
37.8.3	How to Set Database State.....	37-18

Part VII Appendices

A Oracle ADF XML Files

A.1	Introduction to the ADF Metadata Files.....	A-1
A.2	ADF File Overview Diagram	A-2
A.2.1	Oracle ADF Data Control Files	A-2
A.2.2	Oracle ADF Data Binding Files.....	A-3

A.2.3	Web Configuration Files	A-3
A.3	adfm.xml	A-4
A.4	<i>modelProjectName.jpx</i>	A-4
A.5	bc4j.xcfg	A-6
A.6	DataBindings.cpx	A-7
A.6.1	DataBindings.cpx Syntax.....	A-8
A.6.2	DataBindings.cpx Sample.....	A-9
A.7	<i>pageNamePageDef.xml</i>	A-10
A.7.1	PageDef.xml Syntax.....	A-10
A.8	adf-config.xml.....	A-23
A.9	task-flow-definition.xml	A-23
A.10	adf-config.xml.....	A-23
A.11	adf-settings.xml.....	A-24
A.12	web.xml	A-24
A.13	logging.xml	A-25

B Oracle ADF Binding Properties

B.1	EL Properties of Oracle ADF Bindings	B-1
-----	--	-----

C Oracle ADF Permission Grants

C.1	Grantable Actions of ADF Components.....	C-1
-----	--	-----

D ADF Equivalents of Common Oracle Forms Triggers

D.1	Validation & Defaulting (Business Logic)	D-1
D.2	Query Processing	D-2
D.3	Database Connection	D-3
D.4	Transaction "Post" Processing (Record Cache)	D-3
D.5	Error Handling	D-4

E Most Commonly Used ADF Business Components Methods

E.1	Most Commonly Used Methods in the Client Tier	E-1
E.1.1	ApplicationModule Interface	E-1
E.1.2	Transaction Interface	E-2
E.1.3	ViewObject Interface	E-3
E.1.4	RowSet Interface	E-4
E.1.5	RowSetIterator Interface	E-5
E.1.6	Row Interface.....	E-7
E.1.7	StructureDef Interface	E-7
E.1.8	AttributeDef Interface	E-7
E.1.9	AttributeHints Interface.....	E-8
E.2	Most Commonly Used Methods In the Business Service Tier	E-9
E.2.1	Controlling Custom Java Files For Your Components.....	E-9
E.2.2	ApplicationModuleImpl Class.....	E-10
E.2.2.1	Methods You Typically Call on ApplicationModuleImpl.....	E-10
E.2.2.2	Methods You Typically Write in Your Custom ApplicationModuleImpl Subclass...	
	E-10	

E.2.2.3	Methods You Typically Override in Your Custom ApplicationModuleImpl Subclass	E-11
E.2.3	DBTransactionImpl2 Class	E-12
E.2.3.1	Methods You Typically Call on DBTransaction.....	E-13
E.2.3.2	Methods You Typically Override in Your Custom DBTransactionImpl2 Subclass ...	E-13
E.2.4	EntityImpl Class.....	E-14
E.2.4.1	Methods You Typically Call on EntityImpl.....	E-14
E.2.4.2	Methods You Typically Write in Your Custom EntityImpl Subclass	E-15
E.2.4.3	Methods You Typically Override on EntityImpl.....	E-15
E.2.5	EntityDefImpl Class	E-17
E.2.5.1	Methods You Typically Call on EntityDefImpl	E-17
E.2.5.2	Methods You Typically Write on EntityDefImpl.....	E-17
E.2.5.3	Methods You Typically Override on EntityDefImpl	E-18
E.2.6	ViewObjectImpl Class	E-18
E.2.6.1	Methods You Typically Call on ViewObjectImpl.....	E-18
E.2.6.2	Methods You Typically Write in Your Custom ViewObjectImpl Subclass	E-19
E.2.6.3	Methods You Typically Override in Your Custom ViewObjectImpl Subclass	E-20
E.2.7	ViewRowImpl Class	E-20
E.2.7.1	Methods You Typically Call on ViewRowImpl.....	E-20
E.2.7.2	Methods You Typically Write on ViewRowImpl	E-21
E.2.7.3	Methods You Typically Override in Your Custom ViewRowImpl Subclass ...	E-21
E.2.8	Setting Up Your Own Layer of Framework Base Classes	E-22

F ADF Business Components Java EE Design Pattern Catalog

F.1	Java EE Design Patterns Implemented by ADF Business Components	F-1
-----	--	-----

G Performing Common Oracle Forms Tasks in Oracle ADF

G.1	Performing Tasks Related to Data	G-1
G.1.1	How to Retrieve Lookup Display Values for Foreign Keys	G-1
G.1.2	How to Get the Sysdate from the Database	G-2
G.1.3	How to Implement an Isolation Mode That Is Not Read Consistent.....	G-2
G.1.4	How to Implement Calculated Fields.....	G-2
G.1.5	How to Implement Mirrored Items	G-3
G.1.6	How to Use Database Columns of Type CLOB or BLOB	G-3
G.2	Performing Tasks Related to the User Interface	G-3
G.2.1	How to Lay Out a Page	G-3
G.2.2	How to Stack Canvases.....	G-4
G.2.3	How to Implement a Master-Detail Screen.....	G-4
G.2.4	How to Implement an Enter Query Screen.....	G-4
G.2.5	How to Implement an Updatable Multi-Record Table	G-4
G.2.6	How to Create a Popup List of Values	G-4
G.2.7	How to Implement a Dropdown List as a List of Values	G-5
G.2.8	How to Implement a Dropdown List with Values from Another Table	G-5
G.2.9	How to Implement Immediate Locking	G-5
G.2.10	How to Throw an Error When a Record Is Locked	G-5

Preface

Welcome to the *Oracle Fusion Middleware Fusion Developer's Guide for Oracle Application Development Framework*.

Audience

This document is intended for enterprise developers who need to create and deploy database-centric Java EE applications with a service-oriented architecture using the Oracle Application Development Framework (Oracle ADF). This guide explains how to build Fusion web applications using Oracle ADF Business Components, Oracle ADF Controller, Oracle ADF Faces, and JavaServer Faces.

Documentation Accessibility

Our goal is to make Oracle products, services, and supporting documentation accessible, with good usability, to the disabled community. To that end, our documentation includes features that make information available to users of assistive technology. This documentation is available in HTML format, and contains markup to facilitate access by the disabled community. Accessibility standards will continue to evolve over time, and Oracle is actively engaged with other market-leading technology vendors to address technical obstacles so that our documentation can be accessible to all of our customers. For more information, visit the Oracle Accessibility Program Web site at

<http://www.oracle.com/accessibility/>

Accessibility of Code Examples in Documentation

Screen readers may not always correctly read the code examples in this document. The conventions for writing code require that closing braces appear on an otherwise empty line; however, some screen readers may not always read a line of text that consists solely of a bracket or brace.

Accessibility of Links to External Web Sites in Documentation

This documentation may contain links to Web sites of other companies or organizations that Oracle does not own or control. Oracle neither evaluates nor makes any representations regarding the accessibility of these Web sites.

TTY Access to Oracle Support Services

Oracle provides dedicated Text Telephone (TTY) access to Oracle Support Services within the United States of America 24 hours a day, seven days a week. For TTY support, call 800.446.2398.

Related Documents

For more information, see the following documents:

Oracle Fusion Middleware Web User Interface Developer's Guide for Oracle Application Development Framework

Oracle Fusion Middleware Developer's Guide for Oracle Application Development Framework Mobile

Oracle JDeveloper 11g Online Help

Oracle JDeveloper 11g Release Notes, included with your JDeveloper 11g installation, and on Oracle Technology Network

Oracle Fusion Middleware Java API Reference for Oracle ADF Model

Oracle Fusion Middleware Java API Reference for Oracle ADF Controller

Oracle Fusion Middleware Java API Reference for Oracle ADF Lifecycle

Oracle Fusion Middleware Java API Reference for Oracle ADF Faces

Oracle Fusion Middleware Java API Reference for Oracle ADF Share

Oracle Fusion Middleware Java API Reference for Oracle ADF Business Component Browser

Oracle Fusion Middleware Java API Reference for Oracle Generic Domains

Oracle Fusion Middleware interMedia Domains Java API Reference for Oracle ADF Business Components

Oracle Fusion Middleware Tag Reference for Oracle ADF Faces

Conventions

The following text conventions are used in this document:

Convention	Meaning
boldface	Boldface type indicates graphical user interface elements associated with an action, or terms defined in text or the glossary.
<i>italic</i>	Italic type indicates book titles, emphasis, or placeholder variables for which you supply particular values.
<code>monospace</code>	Monospace type indicates commands within a paragraph, URLs, code in examples, text that appears on the screen, or text that you enter.

Part I

Getting Started with Fusion Web Applications

Part I contains the following chapters:

- [Chapter 1, "Introduction to Building Fusion Web Applications with Oracle ADF"](#)
- [Chapter 2, "Introduction to the ADF Sample Application"](#)

Introduction to Building Fusion Web Applications with Oracle ADF

This chapter describes the architecture and key functionality of the Oracle Application Development Framework (Oracle ADF) when used to build a Fusion web application that uses Oracle ADF Business Components, Oracle ADF Model, Oracle ADF Controller, and Oracle ADF Faces Rich Client, along with high-level development practices.

This chapter includes the following sections:

- [Section 1.1, "Introduction to Oracle ADF"](#)
- [Section 1.2, "Oracle ADF Architecture"](#)
- [Section 1.3, "Developing Declaratively with Oracle ADF"](#)
- [Section 1.4, "Working Productively in Teams"](#)
- [Section 1.5, "Learning Oracle ADF"](#)

1.1 Introduction to Oracle ADF

The Oracle Application Development Framework (Oracle ADF) is an end-to-end application framework that builds on Java Platform, Enterprise Edition (Java EE) standards and open-source technologies to simplify and accelerate implementing service-oriented applications. If you develop enterprise solutions that search, display, create, modify, and validate data using web, wireless, desktop, or web services interfaces, Oracle ADF can simplify your job. Used in tandem, Oracle JDeveloper 11g and Oracle ADF give you an environment that covers the full development lifecycle from design to deployment, with drag-and-drop data binding, visual UI design, and team development features built in.

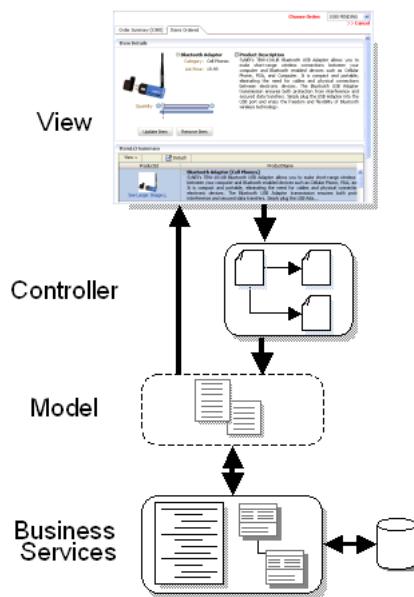
To help illustrate the concepts and procedures in this guide (and other Fusion Middleware developer guides), you can download and view the Fusion Order demo application. The StoreFront module of this application is built using the Fusion web application technology stack, which includes ADF Business Components, ADF Model, ADF Controller, and JavaServer Faces pages with ADF Faces Rich Client components. Screenshots and code samples from this module are used throughout this guide to provide you with real-world examples of using the Oracle ADF technologies in an application that uses the Fusion web technology stack. For more information about the StoreFront module of the Fusion Order Demo application, see [Chapter 2, "Introduction to the ADF Sample Application"](#).

1.2 Oracle ADF Architecture

In line with community best practices, applications you build using the Fusion web technology stack achieve a clean separation of business logic, page navigation, and user interface by adhering to a model-view-controller architecture. As shown in [Figure 1–1](#), in an MVC architecture:

- The model layer represents the data values related to the current page
- The view layer contains the UI pages used to view or modify that data
- The controller layer processes user input and determines page navigation
- The business service layer handles data access and encapsulates business logic

Figure 1–1 MVC Architecture Cleanly Separates UI, Business Logic and Page Navigation

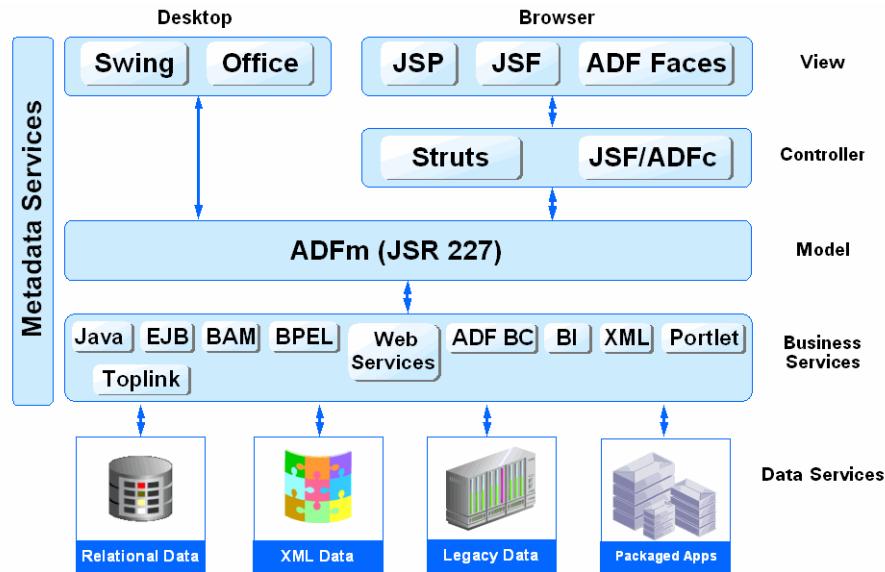


[Figure 1–2](#) illustrates where each ADF module fits in the Fusion web application architecture. The core module in the framework is Oracle ADF Model, a declarative data binding facility that implements the JSR-227 specification. This specification provides an API for accessing declarative data binding metadata. The Oracle ADF Model layer enables a unified approach to bind any user interface to any business service, without the need to write code. The other modules that make up a Fusion web application technology stack are:

- Oracle ADF Business Components, which simplifies building business services.
- Oracle ADF Faces, which offers a rich library of AJAX-enabled UI components for web applications built with JavaServer Faces (JSF).
- Oracle ADF Controller, which integrates JSF with Oracle ADF Model. The ADF Controller extends the standard JSF controller by providing additional functionality, such as reusable task flows that pass control not only between JSF pages, but also between other activities, for instance method calls or other task flows.

Note: In addition to ADF Faces, Oracle ADF also supports using the Swing, JSP, and standard JSF view technologies. For more information about these technologies, refer to the JDeveloper online help.

Figure 1–2 Simple ADF Architecture



1.2.1 ADF Business Components

When building service-oriented Java EE applications, you implement your core business logic as one or more business services. These backend services provide clients with a way to query, insert, update, and delete business data as required while enforcing appropriate business rules. ADF Business Components are prebuilt application objects that accelerate the job of delivering and maintaining high-performance, richly functional, database-centric services. They provide you with a ready-to-use implementation of Java EE design patterns and best practices.

As illustrated in [Figure 1–3](#), Oracle ADF provides the following key components to simplify building database-centric business services:

- Entity object

An entity object represents a row in a database table and simplifies modifying its data by handling all data manipulation language (DML) operations for you. It can encapsulate business logic to ensure that your business rules are consistently enforced. You associate an entity object with others to reflect relationships in the underlying database schema to create a layer of business domain objects to reuse in multiple applications.

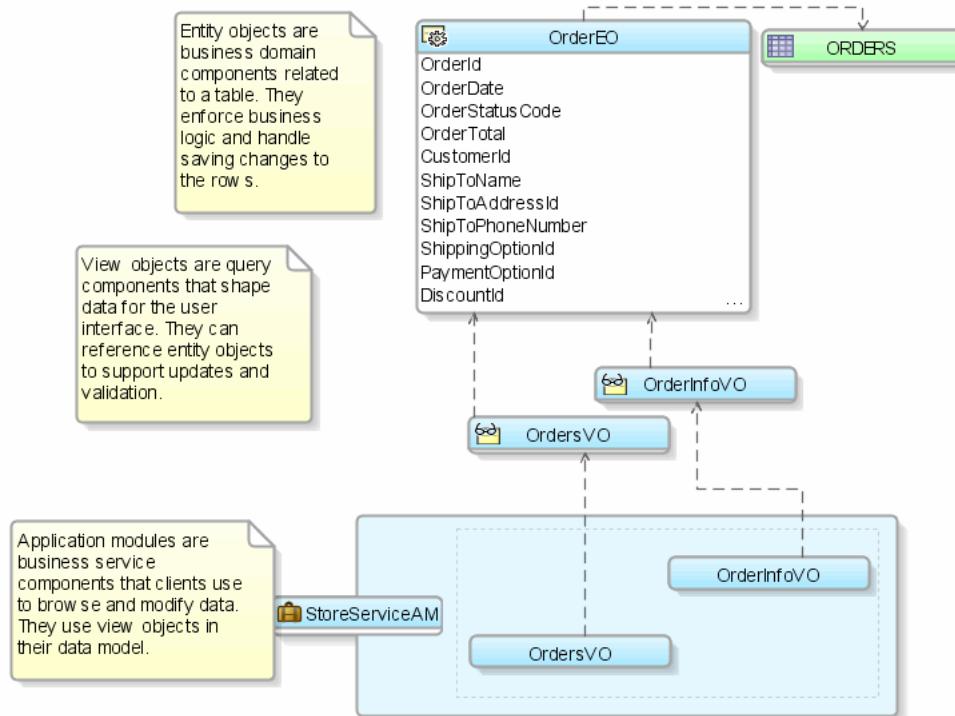
- View object

A view object represents a SQL query and simplifies working with its results. You use the SQL language to join, project, filter, sort, and aggregate data into the shape required by the end-user task being represented in the user interface. This includes the ability to link a view object with other entity objects to create master-detail hierarchies of any complexity. When end users modify data in the user interface, your view objects collaborate with entity objects to consistently validate and save the changes.

- Application module

An application module is the transactional component that UI clients use to work with application data. It defines an updateable data model and top-level procedures and functions (called service methods) related to a logical unit of work related to an end-user task.

Figure 1–3 ADF Business Components Simplify Data Access and Validation



Tip: If you have previously worked with Oracle Forms, note that this combined functionality is the same set of data-centric features provided by the form, data blocks, record manager, and form-level procedures or functions. The key difference in Oracle ADF is that the user interface is cleanly separated from data access and validation functionality. For more information, see [Appendix G, "Performing Common Oracle Forms Tasks in Oracle ADF"](#).

1.2.2 ADF Model Layer

In the model layer, Oracle ADF Model implements the JSR-227 service abstraction called the *data control*. Data controls abstract the implementation technology of a business service by using standard metadata interfaces to describe the service's operations and data collections, including information about the properties, methods, and types involved. Using JDeveloper, you can view that information as icons that you can then drag and drop onto a page. At that point, JDeveloper automatically creates the bindings from the page to the services. At runtime, the ADF Model layer reads the information describing your data controls and data bindings from appropriate XML files and implements the two-way connection between your user interface and your business service.

Oracle ADF provides out-of-the-box data control implementations for the most common business service technologies. Using JDeveloper and Oracle ADF together

provides you a declarative, drag-and-drop data binding experience as you build your user interfaces. Along with support for ADF application modules, ADF Model also provides support for the following service technologies:

- Enterprise JavaBeans (EJB) session beans and JPA entities
- JavaBeans
- Web services
- XML
- CSV files

1.2.3 ADF Controller

In the controller layer, where handling page flow of your web applications is a key concern, ADF Controller provides an enhanced navigation and state management model on top of JSF. JDeveloper allows you to declaratively create task flows where you can pass application control between different types of activities, such as pages, methods on managed beans, declarative case statements, or calls to other task flows.

1.2.4 ADF Faces Rich Client

ADF Faces Rich Client (RC) is a set of standard JSF components that include built-in AJAX functionality. AJAX is a combination of asynchronous JavaScript, dynamic HTML (DHTML), XML, and XMLHttpRequest communication channel. This combination allows requests to be made to the server without fully re-rendering the page. While AJAX allows rich client-like applications to use 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 RC provides over 100 rich components, including hierarchical data tables, tree menus, in-page dialogs, accordions, dividers, and sortable tables. ADF Faces RC also provides ADF Data Visualization components, which are Flash- and SVG-enabled components capable of rendering dynamic charts, graphs, gauges, and other graphics that can provide a real-time view of underlying data. Each component also supports skinning, along with internationalization and accessibility.

To achieve these front-end capabilities, ADF Faces RC components use a rendering kit that handles displaying the component and also provides the JavaScript objects needed for the rich functionality. This built-in support enables you to build rich applications without needing extensive knowledge of the individual technologies on the front or back end.

For more information about ADF Faces RC, including the architecture and detailed information about each of the components, see the *Oracle Fusion Middleware Web User Interface Developer's Guide for Oracle Application Development Framework*.

Along with ADF Faces RC, Oracle ADF also supports the following view technologies:

- Apache MyFaces Trinidad: This is the open source code donation from Oracle to the Apache Software Foundation. ADF Faces RC components are based on these Trinidad components)
- Java Swing and ADF Swing: ADF Swing is the development environment for building Java Swing applications that use the ADF Model layer.
- ADF Mobile: This is a standards-based framework for building mobile applications built on the component model of JSF.

1.3 Developing Declaratively with Oracle ADF

Using JDeveloper 11g with Oracle ADF, you benefit from a high-productivity environment that automatically manages your application's declarative metadata for data access, validation, page control and navigation, user interface design, and data binding.

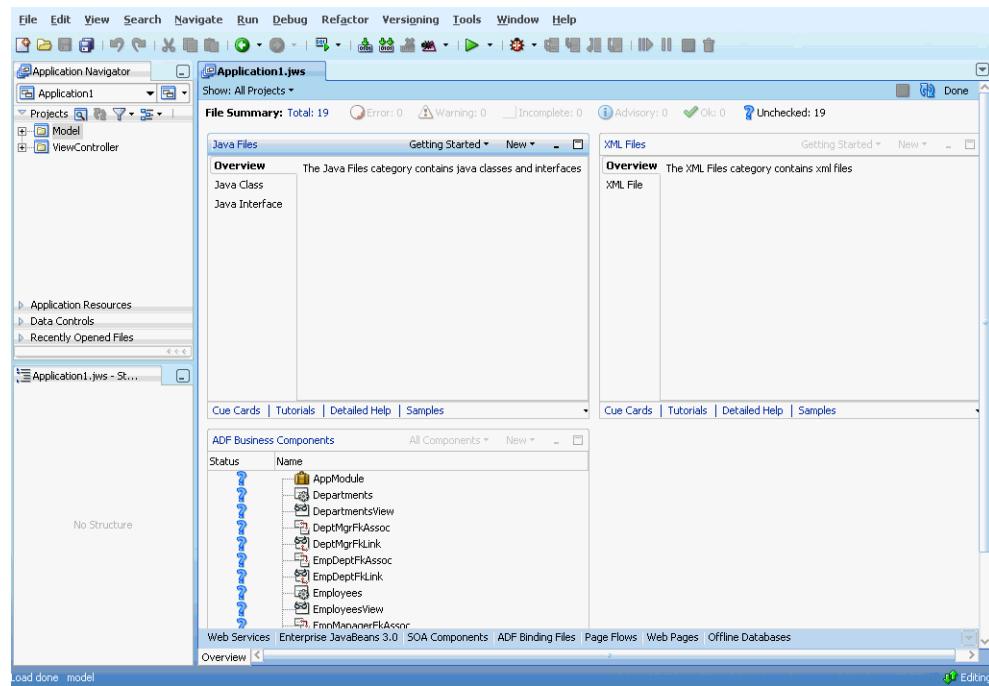
At a high level, the development process for a Fusion web application usually involves the following:

- Creating an application workspace
- Creating use cases
- Designing application control and navigation
- Identifying shared resources
- Creating ADF Business Components to access data
- Implementing the user interface with JSF
- Binding UI components to data using the ADF Model layer
- Incorporating validation and error handling
- Securing the application
- Testing and debugging
- Deploying the application

1.3.1 Creating an Application Workspace

The first step in building a new application is to assign it a name and to specify the directory where its source files will be saved. When you create an application using the application templates provided by JDeveloper, JDeveloper organizes your workspace into projects and creates and organizes many of the configuration files required by the type of application you are creating.

One of these templates is the Fusion Web Application (ADF) template, which provides the correctly configured set of projects you need to create a web application that uses ADF Faces Rich Client for the view, ADF Page Flow for the controller, and ADF Business Components for business services. When you create an application workspace using this template, JDeveloper creates a project named `Model` that will contain all the source files related to the business services in your application, and a project named `ViewController` that will contain all the source files for your ADF Faces view layer, including files for the controller. JDeveloper automatically creates the JSF and ADF configuration files needed for the application. JDeveloper displays all the files for your application in the application overview editor, as shown in [Figure 1–4](#).

Figure 1–4 New Workspace for a Fusion Web Application

JDeveloper also adds the following libraries to the view project:

- JSF 1.2
- JSTL 1.2
- ADF Page Flow Runtime
- ADF Controller Runtime
- ADF Controller Schema
- ADF Faces Runtime 11
- ADF Common Runtime
- ADF Web Runtime
- MDS Runtime
- MDS Runtime Dependencies
- Commons Beutifuls 1.6.1
- Commons Logging 1.0.3
- Commons Collections 2.1
- Trinidad Runtime 11

Once you add a JSF page, JDeveloper adds the following libraries:

- JSP Runtime
- DVT Faces Runtime
- Oracle JEWT

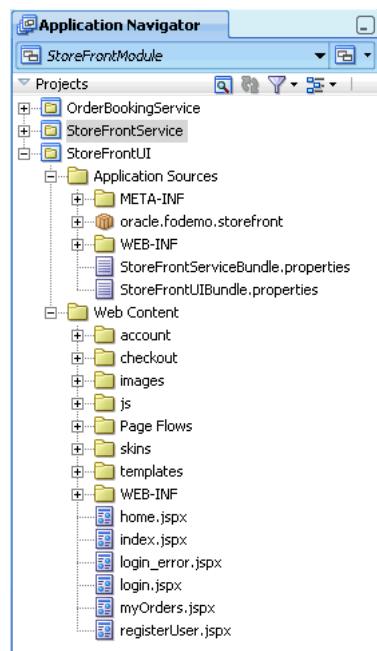
Once the projects are created for you, you can rename them as you need. You can then use JDeveloper to create additional projects, and add the packages and files needed for your application.

Note: If you plan to reuse artifacts in your application (for example, task flows), then you should follow the naming guidelines presented in [Chapter 31, "Reusing Application Components"](#) in order to prevent naming conflicts.

Tip: You can edit the default values used in application templates, as well as create your own templates. To do so, choose **Tools > Manage Application Templates**.

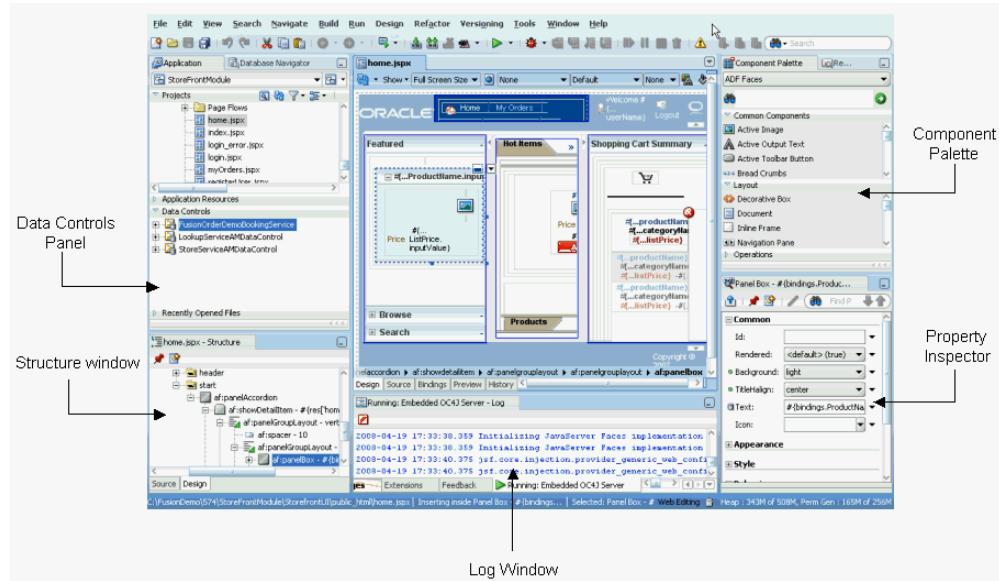
Figure 1–5 shows the different projects, packages, directories, and files for the `StoreFrontModule` application, as displayed in the Application Navigator.

Figure 1–5 StoreFrontModule Application Projects, Packages, and Directories



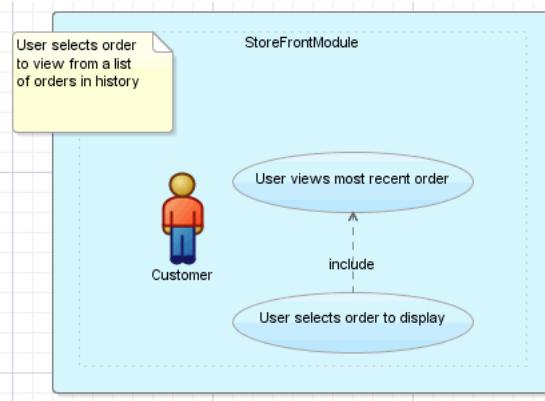
For more information, see "Managing Applications and Projects" in the "JDeveloper Basics" section of the JDeveloper online help.

When you work in your files, you use mostly the editor window, the Structure window, and the Property Inspector, as shown in [Figure 1–6](#). The editor window allows you to view many of your files in a WYSIWYG environment, or you can view a file in an overview editor where you can declaratively make changes, or you can view the source code for the file. The Structure window shows the structure of the currently selected file. You can select objects in this window and then edit the properties for the selection in the Property Inspector.

Figure 1–6 The JDeveloper Workspace

1.3.2 Creating Use Cases

After creating an application workspace, you may decide to begin the development process by doing use case modeling to capture and communicate end-user requirements for the application to be built. [Figure 1–7](#) shows a simple diagram created using the UML modeler in JDeveloper. The diagram represents an end user viewing a list of his orders and then drilling down to view the details of an order. Using diagram annotations, you can capture particular requirements about what end users might need to see on the screens that will implement the use case. For example, in this use case, it is noted that the user will select order details for each order listed.

Figure 1–7 Use Case Diagram for Viewing Order History

For more information about creating use case diagrams, see "Modeling With Diagrams" in the "Designing and Developing Applications" section of the JDeveloper online help.

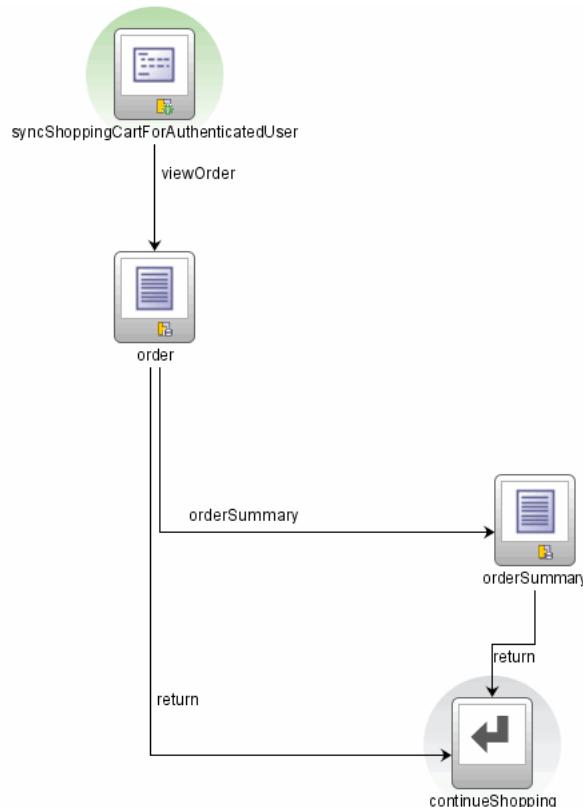
1.3.3 Designing Application Control and Navigation Using ADF Task Flows

By modeling the use cases, you begin to understand the kinds of user interface pages that will be required to implement end-user requirements. At this point, you can begin

to design the flow of your application. In a Fusion web application, you use ADF task flows instead of standard JSF navigation flows. Task flows provide a more modular and transaction-aware approach to navigation and application control. Like standard JSF navigation flows, task flows contain mostly viewable pages. However, instead of describing navigation between pages, task flows facilitate transitions between activities. Aside from navigation, task flows can also have nonvisual activities that can be chained together to declaratively affect the page flow and application behavior. For example, these nonvisual activities can call methods on managed beans, evaluate an EL expression, or call another task flow. This facilitates reuse, as business logic can be invoked independently of the page being displayed.

[Figure 1–8](#) shows the checkout-task-flow task flow from the StoreFront module of the Fusion Order Demo application. In this task flow, `order` and `orderSummary` are view activities that represent pages, while `syncShoppingCartForAuthenticatedUser` is a method call activity. When the user enters this flow, the `syncShoppingCartForAuthenticatedUser` activity is invoked (because it is the entry point for the flow, as denoted by the green circle) and the corresponding method is called. From there, the flow continues to the `order` page. From the `order` page, control can be passed to the `orderSummary` page, or to the `continueShopping` return activity that is the exit point of the flow and passes control back to the home page.

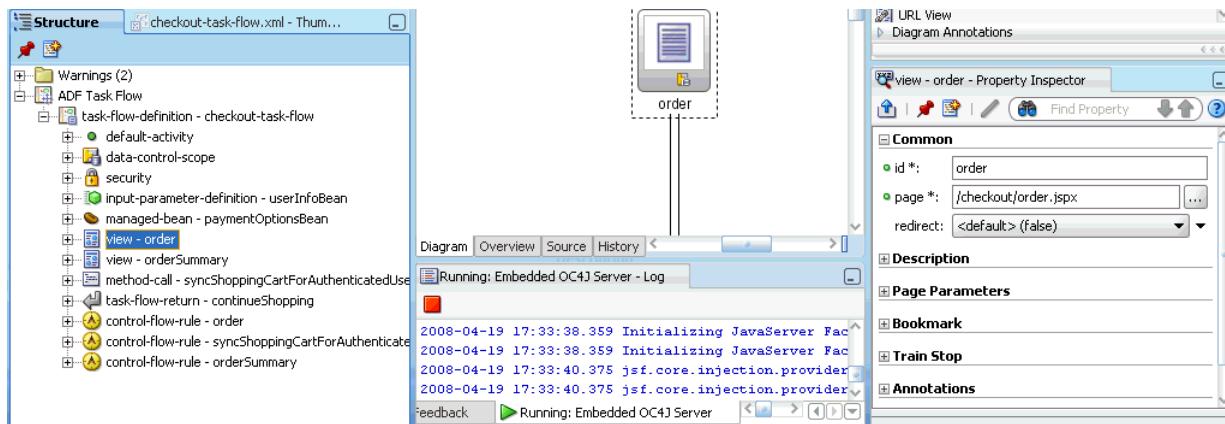
Figure 1–8 Task Flow in the StoreFrontModule Application



The ADF Controller provides a mechanism to declaratively define navigation using control flow rules. The control flow rule information, along with other information regarding the flow, is saved in a configuration file. [Figure 1–9](#) shows the Structure window for the checkout-task-flow task flow. This window shows each of the items configured in the flow, such as the control flow rules. The Property Inspector (by

default, located at the bottom right) allows you to set values for the different elements in the flow.

Figure 1–9 Task Flow Elements in the Structure Window and Property Inspector

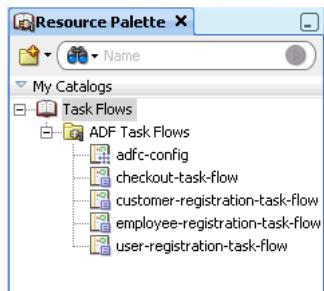


Aside from pages, task flows can also coordinate page fragments. Page fragments are JSF JSP documents that are rendered as content in other JSF pages. You can create page fragments and the control between them in a bounded task flow as you would create pages, and then insert the entire task flow into another page as a region. Because it is simply another task flow, the region can independently execute methods, evaluate expressions, and display content, while the remaining content on the containing page remains the same. For example, before registering a new user, the application needs to determine what kind of user needs to be created. All the logic to do this is handled in the user-registration-task-flow task flow, which is used as a region in the registerUser page.

Regions also facilitate reuse. You can create a task flow as a region, determine the pieces of information required by a task and the pieces of information it might return, define those declaratively as parameters and return values of the region, then drop the region on any page in an application. Depending on the value of the parameter, a different view can display. For information about task flows and creating them, see [Chapter 13, "Getting Started with ADF Task Flows"](#).

1.3.4 Identifying Shared Resources

You may find that some aspects of your application can be reused throughout the application. For example, you may need the functionality of creating an address to appear both when a user registers and when a user creates an order. Or you may find throughout the development process that certain components of your application should be shared throughout the application. You can declaratively create ADF libraries that allow you to package artifacts and reuse them throughout the application. For example, you might create a task flow for the process of creating an address. You can then save this task flow and package it as a library. The library can be sent to other developers who can add it to their a resource catalog, from which they can drag and drop it onto any page where it's needed. [Figure 1–10](#) shows the Resource Palette in JDeveloper.

Figure 1–10 Resource Palette in JDeveloper

When designing the application, be sure to note all the tasks that can possibly become candidates for reuse. For more information about component reuse see [Chapter 31, "Reusing Application Components"](#).

1.3.5 Creating ADF Business Components to Access Data

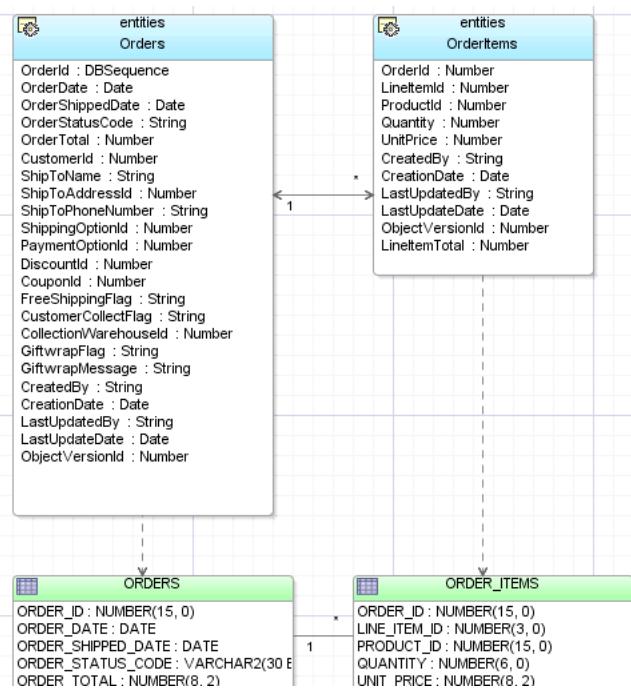
Typically, when you implement business logic as ADF Business Components, you do the following:

- Create entity objects to represent tables that you expect your application to perform a transaction against (if no transaction is to be performed, an entity object is not needed). Add validation and business rules as needed.
- Create view objects that work with the entity objects to query and update the database. These view objects will be used to make the data available for display at your view layer.
- Create the application module that the UI layer of your application will use. This application module contains view object instances in its data model along with any custom methods that users will interact with through the application's web pages.

1.3.5.1 Creating a Layer of Business Domain Objects for Tables

Once you have an understanding of the data that will be presented and manipulated in your application, if you haven't already done so, you can build your database (for more information, see the "Designing Databases" in the "Designing and Developing Applications" section of the JDeveloper online help). Once the database tables are in place, you can create a set of entity objects that represents them and simplifies modifying the data they contain. When you use entity objects to encapsulate data access and validation related to the tables, any pages you build today or in the future that work with these tables are consistently validated. As you work, JDeveloper automatically configures your project to reference any necessary Oracle ADF runtime libraries your application will need at runtime.

For example, the `StoreFrontService` project of the `StoreFrontModule` application contains the business services needed by the application. [Figure 1–11](#) shows two of the entity objects that represent the database tables in that application.

Figure 1–11 Business Components Diagram Showing Entity Objects and Related Tables

To create the business layer, you first create the entity objects based on your database tables. Any relationships between the tables will be reflected as associations between the corresponding entity objects. Alternatively, you can first create the entity objects, and then the associations, and then create database tables from those objects.

Once the entity objects are created, you can define control and attribute hints that simplify the display of the entities in the UI, and you can also add behaviors to the objects. For more information, see [Chapter 4, "Creating a Business Domain Layer Using Entity Objects"](#).

1.3.5.2 Building the Business Services

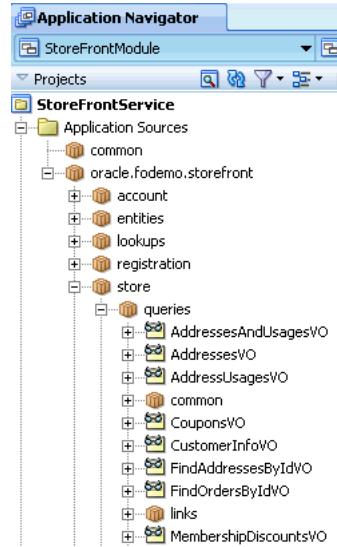
Once the reusable layer of business objects is created, you can implement the application module. An application module is the transactional component that UI clients use to work with application data. It defines an updateable data model and top-level procedures and functions (called *service methods*) for a logical unit of work related to an end-user task.

The application module's data model is composed of instances of the view object components you create that encapsulate the necessary queries. View objects can join, project, filter, sort, and aggregate data into the shape required by the end-user task being represented in the user interface. When the end user needs to update the data, your view objects reference entity objects in your reusable business domain layer. View objects are reusable and can be used in multiple application modules.

When you want the data to display in a consistent manner across all view pages that access that data, you can configure metadata on the view object to determine display properties. The metadata allows you to set display properties in one place and then change them as needed, so that you make the change only in one place instead of on all pages that display the data. Conversely, you can also have the query controlled by the data the page requires. All display functionality is handled by the page. For more information, see [Chapter 5, "Defining SQL Queries Using View Objects"](#).

For example, the `StoreFrontService` project contains the `oracle.fodemo.storefront.store.queries` package, which contains many of the queries needed by the `StoreFrontModule` application, as shown in [Figure 1-12](#).

Figure 1-12 View Objects in the `StoreFrontModule` Application



1.3.5.3 Testing Business Services with the Business Component Browser

While you develop your application, you can iteratively test your business services using the Business Component Browser. The browser allows you to test the queries, business logic, and validation of your business services without having to use or create a user interface or other client to test your services. Using the browser allows you to test out the latest queries or business rules you've added, and can save you time when you're trying to diagnose problems. For more information about developing and testing application modules, see [Chapter 9, "Implementing Business Services with Application Modules"](#).

1.3.6 Implementing the User Interface with JSF

From the task flows you created during the planning stages, you can double-click on the view activity icons to create the actual JSP files. When creating a Fusion web application, you create an XML-based JSP document (which uses the extension `*.jspx`) rather than a `*.jsp` file. Using an XML-based document:

- Simplifies treating your page as a well-formed tree of UI component tags
- Discourages you from mixing Java code and component tags
- Allows you to easily parse the page to create documentation or audit reports
- Allows you to create a view layer that can be personalized

Oracle ADF allows you to create and use page templates. When creating templates, a developer can determine the layout of the page, provide static content that must appear on all pages, and create placeholder attributes that can be replaced with valid values for each page. Each time the template is changed, for example if the layout changes, any page that uses the template will reflect the update.

Most pages in the `StoreFrontModule` application use the `StoreFrontTemplate` template, which provides an area for branding and navigation, a main content area

divided into three panes, and a footer area. If the template designer decides to switch the location of the branding and the navigation, all pages that use the template will automatically reflect that change at runtime.

For more information about creating pages for your Fusion web application, see [Chapter 18, "Getting Started with Your Web Interface"](#). Once your page files are created, you can add databound UI components. [Part IV, "Creating a Databound Web User Interface"](#) is dedicated to creating the view layer of a Fusion web application.

1.3.7 Data Binding with Oracle ADF Model Layer

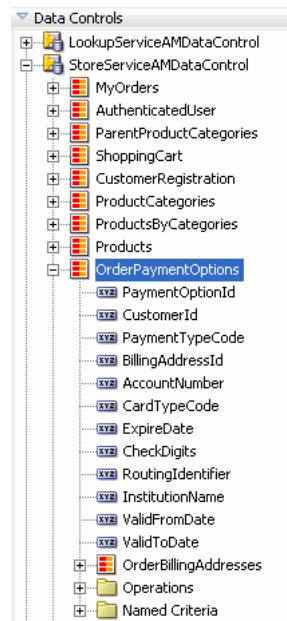
In JSF, you use a simple *expression language* (called EL) to bind to the information you want to present and/or modify. Example expressions look like

`# {UserList.selectedUsers}` to reference a set of selected users, `# {user.name}` to reference a particular user's name, or `# {user.role == 'manager'}` to evaluate whether a user is a manager or not. At runtime, a generic expression evaluator returns the `List`, `String`, and `boolean` value of these respective expressions, automating access to the individual objects and their properties without requiring code.

At runtime, the value of a JSF UI component is determined by its `value` attribute. While a component can have static text as its value, typically the `value` attribute will contain a binding that is an EL expression that the runtime infrastructure evaluates to determine what data to display. For example, an `outputText` component that displays the name of the currently logged-in user might have its `value` attribute set to the expression `# {UserInfo.name}`. Since any attribute of a component can be assigned a value using an EL expression, it's easy to build dynamic, data-driven user interfaces. For example, you could hide a component when a set of objects you need to display is empty by using a boolean-valued expression like `# {not empty UserList.selectedUsers}` in the UI component's `rendered` attribute. If the list of selected users in the object named `UserList` is empty, the `rendered` attribute evaluates to `false` and the component disappears from the page.

In a typical JSF application, you would create objects like the `UserList` object as a managed bean. The JSF runtime manages instantiating these beans on demand when any EL expression references them for the first time. However, in an application that uses the ADF Model layer, instead of binding the UI component attributes to properties or methods on managed beans, JDeveloper automatically binds the UI component attributes to the ADF Model layer, which uses XML configuration files that drive generic data binding features. It implements the two concepts in JSR-227 that enable decoupling the user interface technology from the business service implementation: *data controls* and *declarative bindings*.

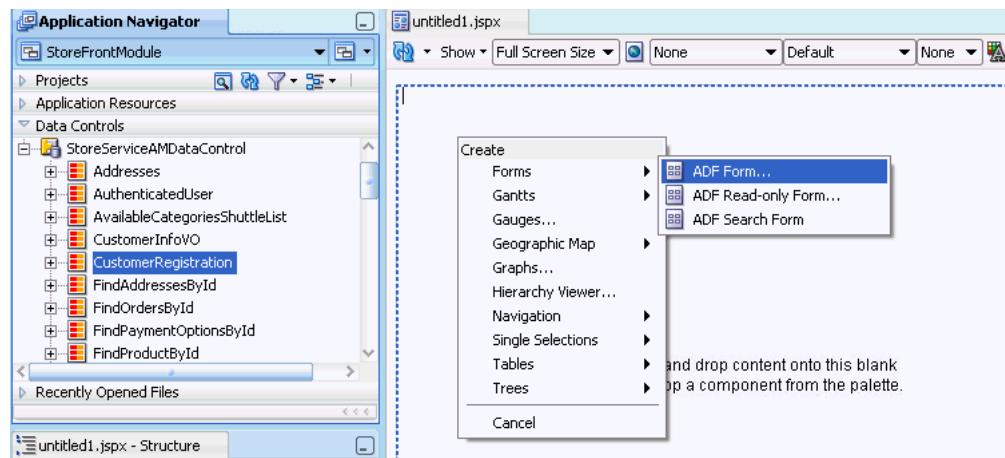
Data controls use XML configuration files to describe a service. At design time, visual tools like JDeveloper can leverage that metadata to allow you to declaratively bind UI components to any data control operation or data collection. For example, [Figure 1-13](#) shows the `StoreServiceAMDataControl` data control as it appears in the Data Controls panel of JDeveloper.

Figure 1–13 StoreFrontServiceAMDataControl

Note that the collections that display in the panel represent the set of rows returned by the query in each view object instance contained in the `StoreServiceAM` application module. For example, the `OrderPaymentOptions` data collection in the Data Controls panel represents the `OrderPaymentOptions` view object instance in the `StoreServiceAM`'s data model. The `OrderBillingAddress` data collection appears as a child, reflecting the master-detail relationship set up while building the business service. The attributes available in each row of the respective data collections appear as child nodes. The data collection level `Operations` node contains the built-in operations that the ADF Model layer supports on data collections, such as `previous`, `next`, `first`, `last`, and so on.

Note: If you create other kinds of data controls for working with web services, XML data retrieved from a URL, JavaBeans, or EJBs, these would also appear in the Data Controls panel with an appropriate display. When you create one of these data controls in a project, JDeveloper creates metadata files that contain configuration information. These additional files do not need to be explicitly created when you are working with Oracle ADF application modules, because application modules are already metadata-driven components, and so contain all the information necessary to be exposed automatically as JSR 227 data controls.

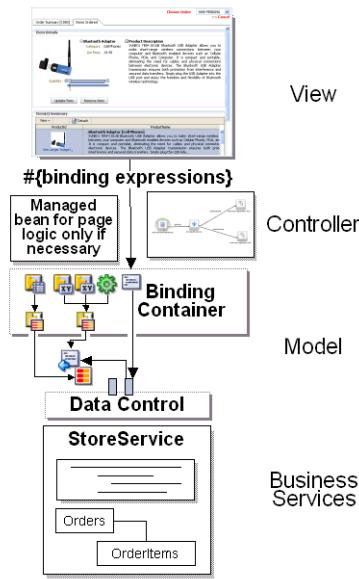
Using the Data Controls panel, you can drag and drop a data collection onto a page in the visual editor, and JDeveloper creates the necessary bindings for you. [Figure 1–14](#) shows the `CustomerRegistration` collection from the `StoreServiceAMDataControl` data control being dragged from the Data Controls panel, and dropped as a form onto a JSF page.

Figure 1–14 Declaratively Creating a Form Using the Data Controls Panel

The first time you drop a databound component from the Data Controls panel on a page, JDeveloper creates an associated page definition file. This XML file describes the group of bindings supporting the UI components on a page. The ADF Model uses this file at runtime to instantiate the page's bindings. These bindings are held in a request-scoped map called the *binding container*. Each time you add components to the page using the Data Controls panel, JDeveloper adds appropriate declarative binding entries into this page definition file. Additionally, as you perform drag-and-drop data binding operations, JDeveloper creates the required tags representing the JSF UI components on the JSF page. For more information about using the Data Controls panel, see [Chapter 11, "Using Oracle ADF Model in a Fusion Web Application"](#).

Note: You can use dynamic UI components that create the bindings at runtime instead of design time. To use dynamic components, you set control hints on your view objects that determine how the data is to be displayed each time the view object is accessed by a page. This ensures that data is displayed consistently across pages, and also allows you to change in a single location, how the data is displayed instead of having to update each individual page. For more information, see [Section 20.7, "Using a Dynamic Form to Determine Data to Display at Runtime"](#).

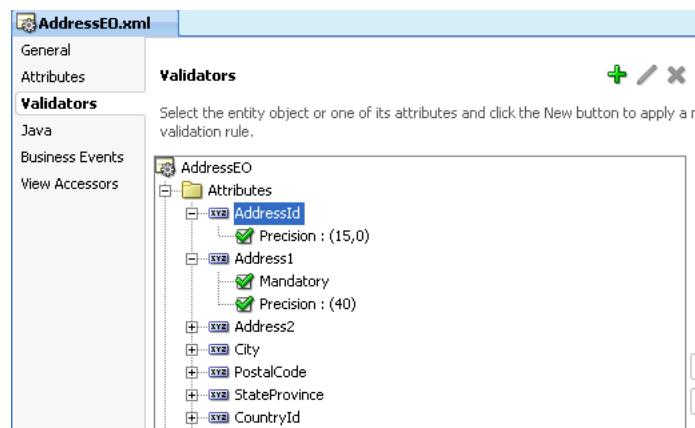
[Figure 1–15](#) illustrates the architecture of a JSF application when you leverage ADF Model for declarative data binding. By combining ADF Model with JSF, you avoid having to write a lot of the typical managed bean code that would be required for real-world applications.

Figure 1–15 Architecture of a JSF Application Using ADF Model Data Binding

Aside from forms and tables that display or update data, you can also create search forms, and databound charts and graphs. For more information about using data controls to create different types of pages, see the chapters contained in [Part IV, "Creating a Databound Web User Interface"](#). For more information about the Data Controls panel and how to use it to create any UI data bound component, see [Chapter 11, "Using Oracle ADF Model in a Fusion Web Application"](#).

1.3.8 Validation and Error Handling

You can add validation to your business objects declaratively using the overview editors for entity and view objects. [Figure 1–16](#) shows the **Validators** tab of the overview editor for the AddressEO entity object.

Figure 1–16 Setting Validation in the Overview Editor

Along with providing the validation rules, you also set the error messages to display when validation fails. To supplement this declarative validation, you can also use Groovy-scripted expressions. For more information about creating validation at the service level, see [Chapter 7, "Defining Validation and Business Rules Declaratively"](#).

Additionally, ADF Faces input components have built-in validation capabilities. You set one or more validators on a component either by setting the `required` attribute or by using the prebuilt ADF Faces validators. You can also create your own custom validators to suit your business needs. For more information, see the "Validating and Converting Input" chapter of the *Oracle Fusion Middleware Web User Interface Developer's Guide for Oracle Application Development Framework*.

You can create a custom error handler to report errors that occur during execution of an ADF application. Once you create the error handler, you only need to register the handler in one of the application's configuration files. For more information, see [Section 26.8, "Customizing Error Handling"](#).

1.3.9 Adding Security

Oracle ADF provides a security implementation that is based on Java Authentication and Authorization Service (JAAS). JAAS is a standard security Application Programming Interface (API) that is added to the Java language through the Java Community Process. It enables applications to authenticate users and enforce authorization. The Oracle ADF implementation of JAAS is role-based. You define these roles and then apply constraints based on these roles throughout the application.

You can also create login pages that allow the application to authenticate users. To do this, you create a managed bean that handles authenticating the user and placing that user information into the session. For more information about securing your application, see [Chapter 28, "Adding Security to a Fusion Web Application"](#).

1.3.10 Testing and Debugging the Web Client Application

Testing an Oracle ADF web application is similar to testing and debugging any other Java EE application. Most errors result from simple and easy-to-fix problems in the declarative information that the application defines or in the EL expressions that access the runtime objects of the page's Oracle ADF binding container. In many cases, examination of the declarative files and EL expressions resolve most problems.

For errors not caused by the declarative files or EL expressions, JDeveloper includes the ADF Logger, which captures runtime trace messages from the ADF Model layer API. The trace includes runtime messages that may help you to quickly identify the origin of an application error. You can also search the log output for specific errors. JDeveloper also includes the ADF Declarative Debugger, a tool that allows you to set breakpoints. When a breakpoint is reached, the execution of the application is paused and you can examine the data that the Oracle ADF binding container has to work with, and compare it to what you expect the data to be. [Chapter 29, "Testing and Debugging ADF Components"](#) contains useful information and tips on how to successfully debug a Fusion web application.

For testing purposes, JDeveloper provides integration with JUnit. You use a wizard to generate regression test cases. For more information, see [Section 29.8, "Regression Testing with JUnit"](#).

1.3.11 Refactoring Application Artifacts

Using JDeveloper, you can easily rename or move the different components in your application. For example, you may find that you need to change the name of your view objects after you have already created them. JDeveloper allows you to easily do this and then propagates the change to all affected metadata XML files. For more information, see [Chapter 30, "Refactoring a Fusion Web Application"](#).

1.3.12 Deploying a Fusion Web Application

You can deploy a Fusion web application to either the integrated WebLogic server within JDeveloper or to a standalone instance. For more information about deployment, see [Chapter 32, "Deploying Fusion Web Applications"](#).

1.4 Working Productively in Teams

Often, applications are built in a team development environment. While a team-based development process follows the development cycle outlined in [Section 1.3, "Developing Declaratively with Oracle ADF"](#), many times developers are creating the different parts of the application simultaneously. Working productively means team members divide the work, understand how to enforce standards, and manage source files with a source control system, in order to ensure efficient application development.

Before beginning development on any large application, a design phase is typically required to assess use cases, plan task flows and screens, and identify resources that can be shared.

The following list shows how the work for a typical Fusion web application might be broken up once an initial design is in place:

- Infrastructure

A DBA creates Ant scripts (or other script files) for building and deploying the finished application. SQL scripts are developed to create the database schema used by the application.

- Entity objects

In a large development environment, a separate development group builds all entity objects for the application. Because the rest of the application depends on these objects, entity objects should be one of the first steps completed in development of the application.

Once the entity objects are finished, they can be shared with other teams using Oracle ADF libraries (see [Section 31.2, "Packaging a Reusable ADF Component into an ADF Library"](#) for more information). The other teams then access the objects by adding them to a catalog in the Resource Palette. In your own application development process, you may choose not to divide the work this way. In many applications, entity objects and view objects might be developed by the same team (or even one person) and would be stored within one project.

- View objects

After the entity objects are created and provided either in a library or within the project itself, view objects can be created as needed to display data (in the case of building the UI) or supply service data objects (when data is needed by other applications in a SOA infrastructure).

When building the Fusion Order Demo application, each developer of a particular page or service was in charge of creating the view objects for that page or service. This was needed because of the tight integration between the view object and its use by a page in the Fusion Order demo; the team who built the UI also built the corresponding view objects.

During development, you may find that two or more view objects are providing the same functionality. In some cases, these view objects can be easily combined by altering the query in one of the objects so that it meets the needs of each developer's page or service.

Once the view objects are in place, you can create the application module, data controls, and add any needed custom methods. The process of creating view objects, reviewing for redundancy, and then adding them to the application module can be an iterative one.

- User interface (UI) creation

With a UI design in place, the view objects in place and the data controls created, the UI can be built either by the team that created the view objects (as described in the previous bullet point) or by a separate team. You can also develop using a UI-first strategy, which would allow UI designers to create pages before the data controls are in place. Oracle ADF provides placeholder data controls that UI designers can use early in the development cycle. For more information, see [Chapter 27, "Designing a Page Using Placeholder Data Controls"](#).

1.4.1 Enforcing Standards

Because numerous individuals divided into separate teams will be developing the application, you should enforce a number of standards before development begins to ensure that all components of the application will work together efficiently. The following are areas within an application where it is important to have standardization in place when working in a team environment:

- Code layout style

So that more than one person can work efficiently in the code, it helps to follow specific code styles. JDeveloper allows you to choose how the built-in code editor behaves. While many of the settings affect how the user interacts with the code editor (such as display settings), others affect how the code is formatted. For example, you can select a code style that determines things like the placement of opening brackets and the size of indents. You can also import any existing code styles you may have, or you can create your own and export them for use by the team. For more information, see "Customizing the Source Editor Environment" in the "JDeveloper Basics" section of the JDeveloper online help.

- Package naming conventions

You should determine not only how packages should be named, but also the granularity of how many and what kinds of objects will go into each package. For example, all managed beans in the StoreFront module are in the `view.managed` package. All beans that contain helper-type methods accessed by other beans are in `util` packages (one for Oracle ADF and one for JSF). All property files are in the `common` package.

- Pages

You can create templates to be used by all developers working on the UI, as described in [Section 1.3.6, "Implementing the User Interface with JSF"](#). This not only ensures that all pages will have the same look and feel, but also allows you to make a change in the template and have the change appear on all pages that use it. For more information, see [Section 18.2, "Using Page Templates"](#).

Aside from using templates, you should also devise a naming standard for pages. For example, you may want to have names reflect where the page is used in the application. To achieve this goal, you can create subdirectories to provide a further layer of organization.

- Connection names: Unlike most JDeveloper and Oracle ADF objects that are created only once per project and by definition have the same name regardless of who sees or uses them, database connection names might be created by individual

team members, even though they map to the same connection details. Naming discrepancies may cause unnecessary conflicts. Team members should agree in advance on common, case-sensitive connection names that should be used by every member of the team.

1.4.2 Using a Source Control System

When working in a team environment, you will need to use a source control system. By default, Oracle JDeveloper provides integrated support for both the CVS and Subversion source control systems, though others may be available through extensions. You can also create an extension that allows you to work with another system in JDeveloper. For information about using these systems within JDeveloper, see the "Using Versioning" topic in the "Designing and Developing Applications" section of the JDeveloper online help.

Following are suggestions for using source control with a Fusion web application:

- Checking out files

Using JDeveloper, you can create a connection to the source control server and use the source control window to check out the source. When you work locally in the files, the pending changes window notifies you of any changed files. You can create a script using Apache Ant (which is integrated into JDeveloper). You can then use the script to build all application workspaces locally. This can ensure that the source files compile before you check the changed files into the source control repository. To find out how to use Apache Ant to create scripts, see "Building With Apache Ant" in the "Designing and Developing Applications" section of the JDeveloper online help.

- Automating builds

Consider running a continuous integration tool. Once files are checked into the source server, the tool can be used to recognize either that files have changed or to check for changed files at determined intervals. At that point, the tool can run an Ant script on the server that copies the full source (note that this should be a copy, and not a checkout), compiles those files, and if the compilation is successful, creates a zip file for consumers (not developers) of the application to use. The script should then clean up the source directory. Running a continuous integration tool will improve confidence in the quality of the code in the repository, encourage developers to update more often, and lead to smaller updates and fewer conflicts. Examples of continuous integration tools include Apache Gump and Cruise Control.

- Updating and committing files

When working with Subversion, updates and commits should be done at the Working Copy level, not on individual files. If you attempt to commit and update an individual file, there is a chance you will miss a supporting metadata file and thereby corrupt the committed copy of the application.

- Resolving merge conflicts

When you add or remove business components in a JDeveloper ADF Business Components project, JDeveloper reflects it in the project file (.jpr). When you create (or refactor) a component into a new package, JDeveloper reflects that in the project file and in the business components project file (.jspx). Although the XML format of these project control files has been optimized to reduce occurrences of merge conflicts, merge conflicts may still arise and you will need to resolve them using JDeveloper's **Resolve Conflicts** option on the context menu of each affected file.

After resolving merge conflicts in any ADF Business Components XML component descriptor files, the project file (.jpr) for an ADF Business Components project, or the corresponding business components project file (.jspx), close and reopen the project to ensure that you're working with latest version of the component definitions. To do this, select the project in the Application Navigator, choose **File > Close** from the JDeveloper main menu, and then expand the project again in the Application Navigator.

1.5 Learning Oracle ADF

In addition to this developers guide, Oracle also offers the following resources to help you learn how you can best use Oracle ADF in your applications:

- Cue Cards in JDeveloper: JDeveloper cue cards provide step-by-step support for the application development process using Oracle ADF. They are designed to be used either with the included examples and a sample schema, or with your own data. Cue cards also include topics that provide more detailed background information, viewlets that demonstrate how to complete the steps in the card, and code samples. Cue cards provide a fast, easy way to become familiar with the basic features of Oracle ADF, and to work through a simple end-to-end task.
- Tutorials on Oracle Technology Network: These short tutorials allow you to gain valuable hands-on experience, and then use the lessons as the foundation for your own implementation.
- *Oracle Fusion Middleware Web User Interface Developer's Guide for Oracle Application Development Framework*: Provides detailed information, procedures, and practices for using the ADF Faces Rich Client components and architecture.

Introduction to the ADF Sample Application

As a companion to this guide, the StoreFront module of the Fusion Order Demo application was created to demonstrate the use of the Fusion web application technology stack to create transaction-based web applications as required for a web shopping storefront. The demonstration application is used as an example throughout this guide to illustrate points and provide code samples.

Before examining the individual components and their source code in depth, you may find it helpful to install and become familiar with the functionality of the Fusion Order Demo application.

This chapter includes the following sections:

- [Section 2.1, "Introduction to the Oracle Fusion Order Demo"](#)
- [Section 2.2, "Setting Up the Fusion Order Demo Application"](#)
- [Section 2.3, "Taking a Look at the Fusion Order Demo Application"](#)

2.1 Introduction to the Oracle Fusion Order Demo

In this sample application, electronic devices are sold through a storefront-type web application. Customers can visit the web site, register, and place orders for the products. In order to register customers and fulfill orders, currently only a single application is in place. In a future release, several applications will cooperate. For a detailed description of how the application works at runtime, see [Section 2.3, "Taking a Look at the Fusion Order Demo Application"](#).

In order to view and run the demo, you need to install Oracle JDeveloper 11g. You then need to download the application for this demonstration. Once the application is installed and running, you can view the code using Oracle JDeveloper. You can view the application at runtime by registering as a new customer (or logging in as an existing customer) and placing an order.

2.2 Setting Up the Fusion Order Demo Application

The Fusion Order Demo application runs using an Oracle database and Oracle JDeveloper 11g. The platforms supported are the same as those supported by JDeveloper.

To prepare the environment and run the Fusion Order Demo application, you must:

1. Install Oracle JDeveloper 11g and meet the installation prerequisites. The Fusion Order Demo application requires an existing Oracle database. For details, see [Section 2.2.1, "How to Download the Application Resources"](#).

2. Install the Fusion Order Demo application from the Oracle Technology Network. For details, see [Section 2.2.2, "How to Install the Fusion Order Demo Application"](#).
3. Install Mozilla FireFox, version 2.0 or higher, or Internet Explorer, version 7.0 or higher.
4. Run the application on a monitor that supports a screen resolution of 1024 X 768 or higher. For details, see [Section 2.2.3, "How to Run the Fusion Order Demo Application"](#).

2.2.1 How to Download the Application Resources

The Fusion Order Demo application requires an existing Oracle database. You run the Fusion Order Demo application using Oracle JDeveloper 11g.

Do the following before installing the Fusion Order Demo application:

- Install Oracle JDeveloper. You need Oracle JDeveloper 11g Studio Edition to view the application's projects and run the application using the JDeveloper integrated server. You can download Oracle JDeveloper from:

<http://www.oracle.com/technology/products/jdev/11/index.html>

- Download the Fusion Order Demo application ZIP file (`FOD_11.zip`). You can download the ZIP file from:

<http://www.oracle.com/technology/products/jdev/samples/fod/index.html>

- Install an Oracle database. The Fusion Order Demo application requires a database for its data.

The SQL scripts were written for an Oracle database, so you will need some version of an Oracle RDBMS, such as 11g, or XE. The scripts will not install into Oracle Lite. If you wish to use Oracle Lite or some other database, then you will need to modify the database scripts accordingly. You can download an Oracle database from:

<http://www.oracle.com/technology/>

Specifically, the small footprint of the Oracle Express Edition (XE) is ideally suited for setting up the database on your local machine. You can download it from:

<http://www.oracle.com/technology/products/database/xe/>

2.2.2 How to Install the Fusion Order Demo Application

You can download the Fusion Order Demo application from the Oracle Technology Network (OTN) web site.

To download and install the demo:

1. Navigate to <http://www.oracle.com/technology/products/jdev/samples/fod/index.html> and download the ZIP file to a local directory.
2. Start Oracle JDeveloper 11g and from the main menu choose **File > Open**.
3. In the Open dialog, browse to the location where you extracted the ZIP file to in Step 1 and select **Infrastructure.jws** from the **Infrastructure** directory. Click **Open**.
4. In the Application Navigator, expand **MasterBuildScript** and then **Resources**, and double-click **build.properties**.

5. In the editor, modify the properties shown in [Table 2–1](#) for your environment.

Table 2–1 Properties Required to Install the Fusion Order Demo Application

Property	Description
jdeveloper.home	The root directory where you have Oracle JDeveloper 11g installed. For example: C:/JDeveloper/11/jdeveloper
jdbc.urlBase	The base JDBC URL for your database in the format jdbc:oracle:thin:@<yourhostname>. For example: jdbc:oracle:thin:@localhost
jdbc.port	The port for your database. For example: 1521
jdbc.sid	The SID of your database. For example: ORCL or XE
db.adminUser	The administrative user for your database. For example: system
db.demoUser.tablespace	The table space name where FOD users will be installed. For example: USERS

6. From the JDeveloper main menu, choose **File > Save All**.
7. In the Application Navigator, under the **Resources** node, right-click **build.xml** and choose **Run Ant Target > buildAll**.

This creates the FOD user and populates the tables in the FOD schema. In the Apache Ant - Log, you will see a series of SQL scripts and finally:

```
buildAll:
BUILD SUCCESSFUL
Total time: nn minutes nn seconds
```

For more information on the demo schema and scripts, see the `README.txt` file in the `MasterBuildScript` project.

2.2.3 How to Run the Fusion Order Demo Application

The Fusion Order Demo application consists of a web user interface and a business components layer. Specifically, the following projects are part of the Fusion Order Demo application:

- **StoreFrontService**: Provides access to the storefront data and provides transaction support to update data for customer information and orders.
- **StoreFrontUI**: Provides web pages that the customer uses to browse the storefront, place orders, register on the site, view order information, and update the user profile.

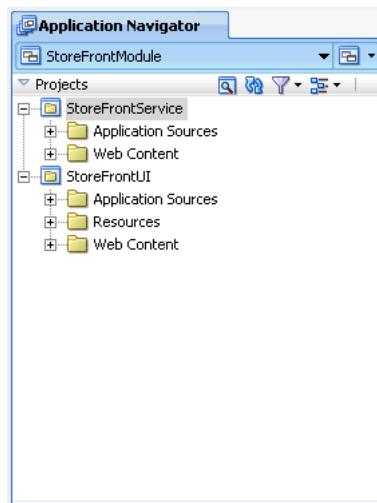
You run the Fusion Order Demo application by running the `home.jsp` page in the `StoreFrontUI` project. The `StoreFrontUI` project uses JavaServer Faces (JSF) as the view technology, and relies on the Oracle ADF Model layer to interact with Oracle ADF Business Components in the `StoreFrontService` project. To learn more about the Fusion Order Demo application and to understand its implementation details, see [Section 2.3, "Taking a Look at the Fusion Order Demo Application"](#).

To run the Fusion Order Demo application:

1. Open the application in Oracle JDeveloper:
 - a. From the JDeveloper main menu, choose **File > Open**.
 - b. Navigate to the location where you extracted the demo ZIP file to and select the **StoreFrontModule.jws** application workspace from the **StoreFrontModule** directory. Click **Open**.

Figure 2–1 shows the Application Navigator after you open the file for the application workspace. For a description of each of the projects in the workspace, see [Section 2.3, "Taking a Look at the Fusion Order Demo Application"](#).

Figure 2–1 The Fusion Order Demo Projects in Oracle JDeveloper



2. In the Application Navigator, click the Application Resources accordion title to expand the panel.
3. In the Application Resources panel, expand the **Connections** and **Database** nodes.
4. Right-click **FOD** connection and choose **Properties**.
5. In the Edit Database Connection dialog, modify the connection information shown in [Table 2–2](#) for your environment.

Table 2–2 Connection Properties Required to Run the Fusion Order Demo Application

Property	Description
Host Name	The host name for your database. For example: localhost
JDBC Port	The port for your database. For example: 1521
SID	The SID of your database. For example: ORCL or XE

Do not modify the user name and password **fod/fusion**. These must remain unchanged. Click **OK**.

6. In the Application Navigator, right-click **StoreFrontService** and choose **Rebuild**.

7. In the Application Navigator, right-click **StoreFrontUI** and choose **Run StoreFrontUI.jpr**.

The home .jspx page within the **StoreFrontUI** project is the default run target. When you run the default target, JDeveloper will launch the browser and display the Fusion Order Demo application home page.

Once the home page appears, you can browse the web site as an anonymous user, or you can choose to log in as a registered customer. The Fusion Order Demo application ships with predefined customer data. Because the Fusion Order Demo application implements Oracle ADF Security to manage access to Oracle ADF resources, only the authenticated user will be able to view orders in their cart. [Table 2–3](#) shows the preregistered customer that you may use to log in.

Table 2–3 Preregistered Customer in the Fusion Order Demo Application

Username	Password	Application Role	Notes
ngreenbe	welcome1	fod-user	Can add items to cart and view checkout and order information.

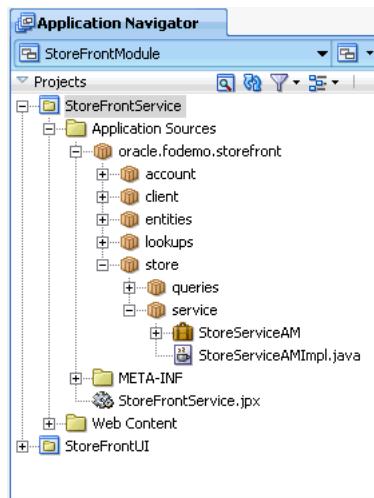
2.3 Taking a Look at the Fusion Order Demo Application

Once you have opened the projects in Oracle JDeveloper, you can then begin to review the artifacts within each project. The development environment for the Fusion Order Demo application is divided into two projects: the **StoreFrontService** project and the **StoreFrontUI** project.

Note: In this JDeveloper release, the Fusion Order Demo application currently has these limitations:

- The application supports the new customer registration process, but that user is not added to the security implementation. Thus, you must use the predefined customer's user name and password to log in.
 - The application does not provide support for order processing in this release. In the R1 release, creating an order will invoke the ESB that then invokes the BPEL process flow that handles the order. In this release, orders are created in the database but no further processing is handled.
-

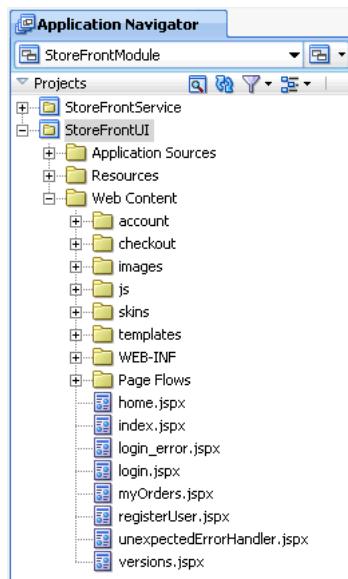
The **StoreFrontService** project contains the classes that allow the product data to be displayed in the web application. [Figure 2–2](#) shows the **StoreFrontService** project and its associated directories.

Figure 2–2 The StoreFrontService Project

The **StoreFrontService** project contains the following directories:

- **Application Sources:** Contains the files used to access the product data. Included are the metadata files used by Oracle Application Development Framework (Oracle ADF) to bind the data to the view.
- **Web Content:** Contains a file used in deployment.

The **StoreFrontUI** project contains the files for the web interface, including the backing beans, deployment files, and JSPX files. [Figure 2–3](#) shows the **StoreFrontUI** project and its associated directories.

Figure 2–3 The StoreFrontUI Project

The **StoreFrontUI** project contains the following directories:

- **Application Sources:** Contains the code used by the web client, including the managed and backing beans, property files used for internationalization, and the metadata used by Oracle ADF to display bound data.

- Resources: Contains the files used to deploy the application.
- Web Content: Contains the web files, including the JSP files, images, skin files, deployment descriptors, and libraries.

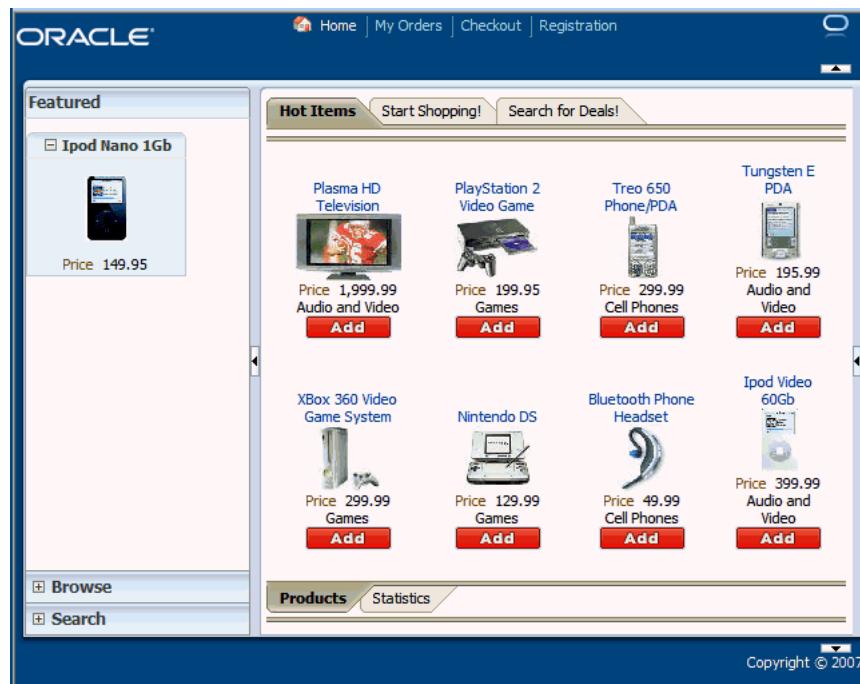
2.3.1 Anonymous Browsing

You start the Fusion Order Demo application by running the `home.jsp` page in the `StoreFrontUI` project. For details about running the application using the default target, `home.jsp` page, see [Section 2.2.3, "How to Run the Fusion Order Demo Application"](#).

When you enter the storefront site, the site is available for anonymous browsing. You can use this page to browse the catalog of products without logging into an account. The initial view shows the featured products that the site wishes to promote and gives you access to the full catalog of items. Products are presented as images along with the name of the product. Page regions divide the product catalog area from other features that the site offers.

[Figure 2-4](#) shows the home page.

Figure 2-4 Home Page with Multiple Regions



Where to Find Implementation Details

Following are the sections of the *Oracle Fusion Middleware Fusion Developer's Guide for Oracle Application Development Framework* that describe how to create a databound web page:

- Providing the structure for the web page

The home page separates features of the site into regions that are implemented using a combination of Oracle ADF Faces (ADF Faces) templates and JavaServer Faces (JSF) page fragments. ADF Faces templates and the fragments allow you to add ADF databound components. For information about the steps you perform

before adding databound user interface components to a web page, see [Section 18.1, "Introduction to Developing a Web Application with ADF Faces"](#).

- **Displaying information on a web page**

To support data binding, the featured items on the tabbed region of the home page use EL (Expression Language) expressions to reference ADF data control usages in the declarative ADF page definition file. The page definition file, which JDeveloper creates for you when you work with the Data Controls panel to drag and drop databound ADF Faces components, is unique to each web page or page fragment. The ADF data control usages enable queries to the database and ultimately work with the JSF runtime to render the databound ADF Faces components, such as the ADF Faces image component used to display images from the PRODUCT_IMAGES table. For information about creating a databound web page that references the ADF page definition file, see [Section 20.1, "Introduction to Creating a Basic Databound Page"](#).

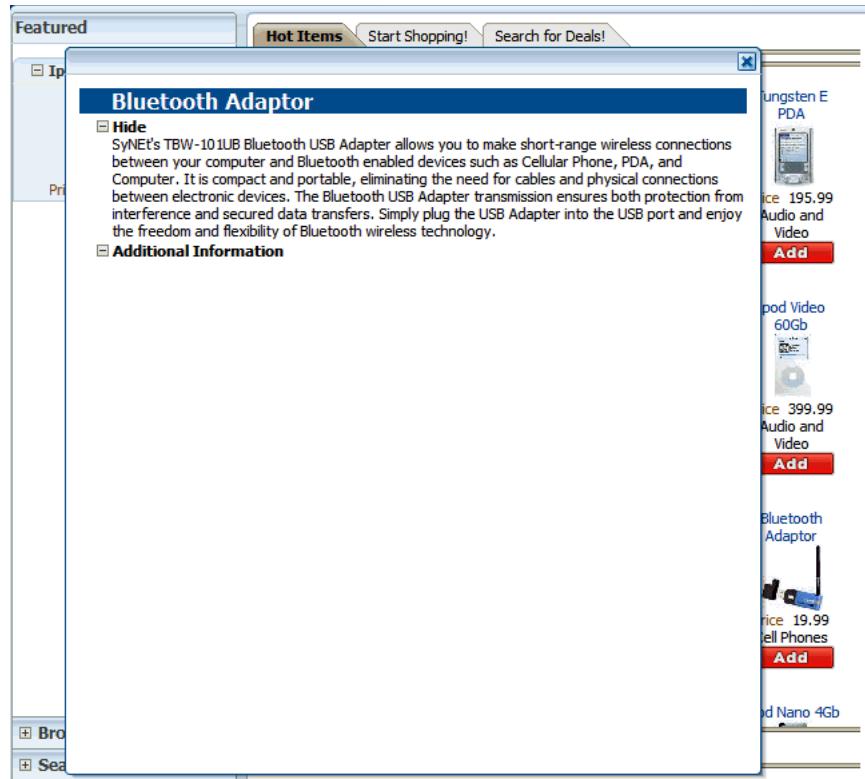
- **Managing entry points to the application**

The home page is supported by an ADF unbounded task flow. In general, the Fusion web application relies on this Oracle ADF Controller feature to define entry points to the application. The unbounded task flow for the entire home page and its page fragments describes view activities for displaying the home page, displaying the orders page, displaying the register user page, and it defines a task flow reference to manage the checkout process. JDeveloper helps you to create the task flow with visual design elements that you drag and drop from the Component Palette. When you create an unbounded task flow, the elements allow you to identify how to pass control from one activity in the application to the next. Because a view activity must be associated with a web page or page fragment, JDeveloper allows you also to create the files for the web page or fragment directly from the task flow diagram. The process of creating a task flow adds declarative definitions to an ADF task flow configuration file. The resulting diagram lets you work with a visual control flow map of the pages and referenced task flows for your application. For more information about specifying the entry points to the application using an ADF unbounded task flows, see [Section 13.1, "Introduction to ADF Task Flows"](#).

2.3.1.1 Viewing Product Details

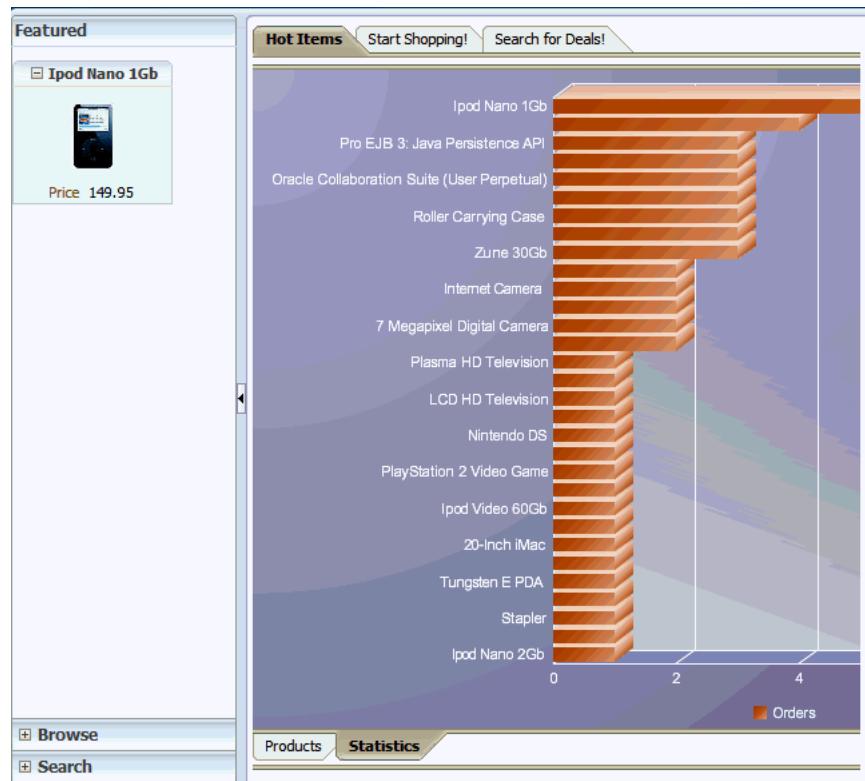
To view detailed product information, you can click the product name link for any product in the home page. The product information is laid out with collapsing nodes organized by categories.

[Figure 2–5](#) shows the detail dialog that you can view for a product.

Figure 2–5 Home Page - Product Details Popup

You can also select the **Statistics** subtab on the home page to view a graphical representation of the number of orders that customers have placed for the featured items. To present the information so that quantities are easily compared, the graph sorts the products by the number of items ordered, in descending order.

Figure 2–6 shows the bar graph used to display the featured products' current order details.

Figure 2–6 Home Page - Statistics for Featured Items

Where to Find Implementation Details

Following are the sections of the *Oracle Fusion Middleware Fusion Developer's Guide for Oracle Application Development Framework* that describe how to develop the components used to support browsing product details:

- Triggering an action to display data

To display data from the data model, user interface components in the web page are bound to ADF Model layer binding objects using JSF Expression Language (EL) expressions. For example, when the user clicks on a link to display an informational dialog, the JSF runtime evaluates the EL expression for the dialog's UI component and pulls the value from the ADF Model layer. At design time, JDeveloper creates declarative definitions of these ADF Model runtime objects when you work with the Data Controls panel to drag an attribute for an item of a data collection, and then choose to display the value, for example, as either read-only text or as an input text field with a label. JDeveloper creates all the necessary JSF tag and binding code needed to display and update the associated data. For more information about the Data Controls panel and the declarative binding experience, see [Section 11.1, "Introduction to ADF Data Binding"](#).

- Displaying data in graphical format

JDeveloper allows you to create databound components declaratively for your JSF pages, meaning you can design most aspects of your pages without needing to look at the code. By dragging and dropping items from the Data Controls panel, JDeveloper declaratively binds ADF Faces UI components and ADF Data Visualization graph components to attributes on a data control using an ADF binding. For more information, see [Section 24.1, "Introduction to Creating ADF Data Visualization Components"](#).

2.3.1.2 Browsing the Product Catalog

To begin browsing, click the **Start Shopping** tab in the home page. This action changes the region of the page used to display details about featured products to a region that displays a product categories tree. You can collapse and expand the branch nodes of the tree to view the various product categories that make up the product catalog. The tree displays the product categories in alphabetical order, by category names. When you want to view all the products in a particular category, click its category node in the tree (for example, click **Electronics**, **Media**, or **Office**). The site refreshes the product information region to display the list of products organized as they appear in the database with an image and an accompanying description.

Figure 2–7 shows the home page with all the products in the **Electronics** category displayed.

Figure 2–7 Home Page - Product Categories View



Where to Find Implementation Details

Following are the sections of the *Oracle Fusion Middleware Fusion Developer's Guide for Oracle Application Development Framework* that describe how to implement master-detail navigation using separate components:

- Synchronizing the display of data for separate components

You can declaratively create pages that display master-detail data using the Data Controls panel. The Data Controls panel displays master-detail related objects in a hierarchy that mirrors the one you defined in the application module data model, where the detail objects are children of the master objects. The Data Controls panel provides prebuilt master-detail ADF Faces components that display both the master and detail objects on the same page as any combination of read-only tables and forms. All you have to do is drop the detail collection on the page and choose the type of widget you want to use. If you do not want to use the prebuilt

master-detail components, you can drag and drop master and detail objects individually as tables and forms on a single page or on separate pages. For more information about the data model, see [Section 3.4, "Overview of the UI-Aware Data Model"](#). For more information about various types of pages that display master-detail related data, see [Section 22.1, "Introduction to Displaying Master-Detail Data"](#).

- Sorting data that displays in tables

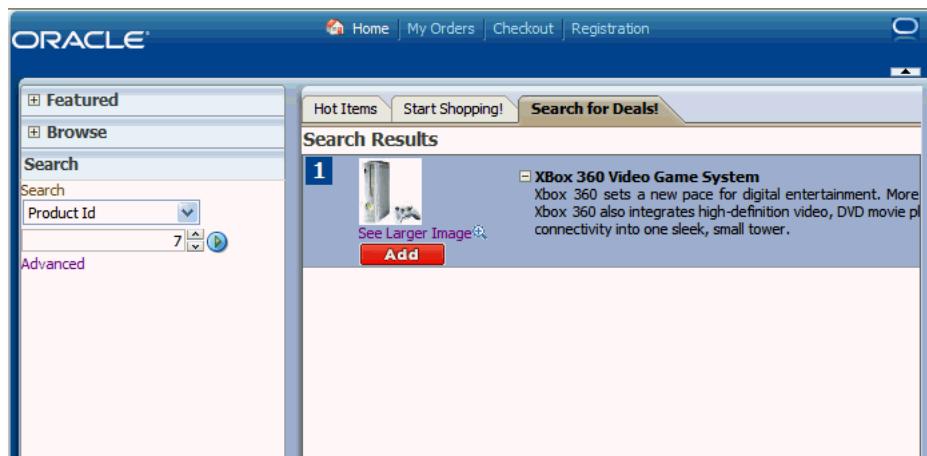
Unlike with forms where you bind the individual UI components that make up a form to the individual attributes on the data collection, you bind the ADF Faces table component to the complete collection or to a range of data objects from the collection. The specific components that display the data in the columns are then bound to the attributes of the collection. The iterator binding handles displaying the correct data for each object, while the table component handles displaying each object in a row. You can set the **Sort** property for any column when you want the iterator to perform an order-by query to determine the order. You can also specify an **ORDER BY** clause for the query that the view object in the data model project defines. For more information about binding table components to a collection, see [Section 21.1, "Introduction to Adding Tables"](#). For more information about creating queries that sort data in the data model, see [Section 5.2, "Populating View Object Rows from a Single Database Table"](#).

2.3.1.3 Searching for Products

To search the product catalog, you have several choices. You can begin either by clicking the selection icon (a + symbol) on the **Search** tab on the accordion panel or by clicking the **Search for Deals** tab in the main region. When you click either of these, the home page displays both regions at once to allow you to enter a search criteria and view the search results. You use the **Search** tab on the accordion panel to perform a simple keyword search against the attributes common to all products, such as product names or product descriptions. When you select the attribute to search on from the dropdown list, the panel renders a search field using an appropriate input component to accept the search criteria. For example, in the case of the default searchable attribute **ProductId**, the search field uses a scrollable selection list field to increment the numeric value.

[Figure 2–8](#) shows the home page with the search results returned for the product with an ID equal to 7.

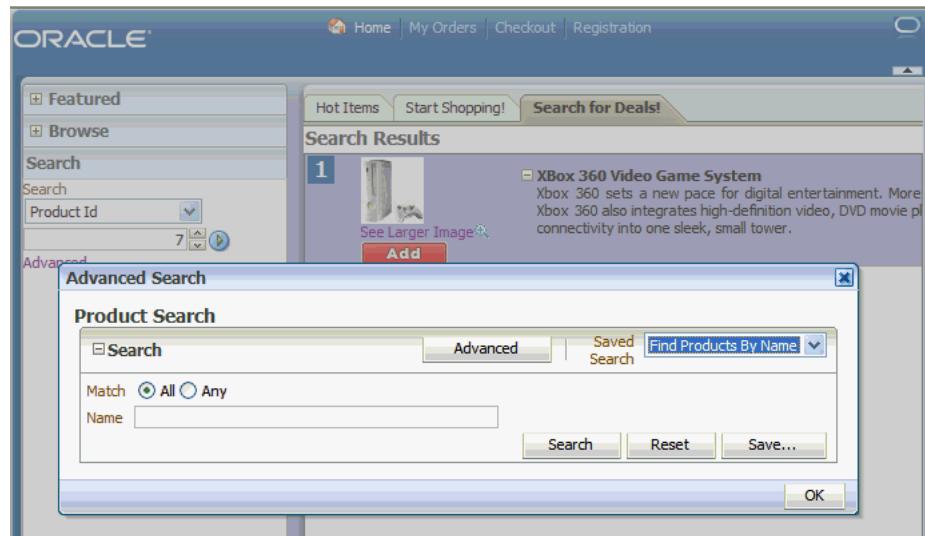
Figure 2–8 Home Page - Search View



As an alternative to entering a simple search, you can use the advanced search feature to define and save search criteria based on any combination of searchable fields that you select for the product. Click the **Advanced** link to open the Advanced Search dialog. Developer-defined saved searches like **Find Products By Name** appear in the **Saved Search** dropdown list.

Figure 2–9 shows the Advanced Search dialog with a single search criteria, **Name**, that the **Find Products By Name** saved search defines.

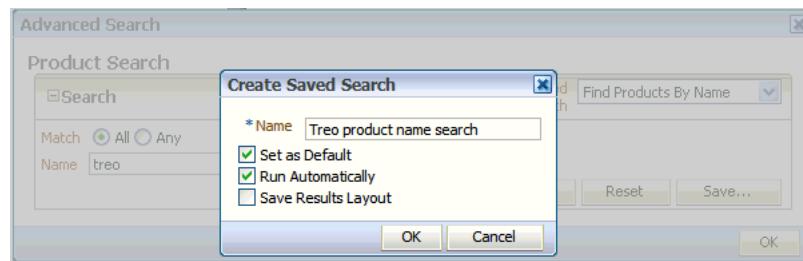
Figure 2–9 Home Page - Advanced Search Dialog



In addition to the developer-defined saved searches available in the Advanced Search dialog, the end user can create saved searches that will persist for the duration of their session. Enter the product search criteria in the Advanced Search dialog, then click the **Save** button to open the Create Saved Search dialog.

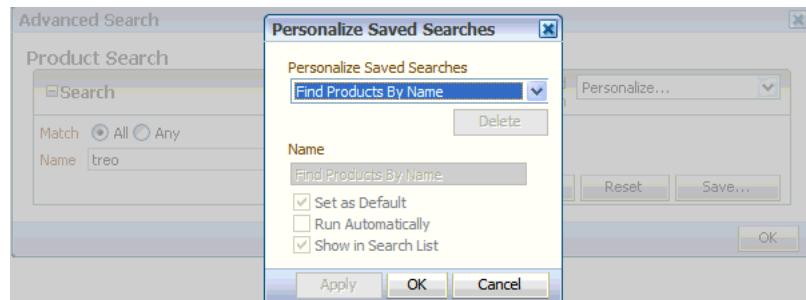
Figure 2–10 shows the Create Saved Search dialog that you use to specify how you want to save the search criteria you entered in the Advanced Search dialog. You can name the search, for example, **Treo product name search**, so that it will display in the **Saved Search** dropdown list of the Advanced Search dialog.

Figure 2–10 Home Page - Advanced Search Dialog - Saved Searches Option



You can also manage your saved searches by selecting the **Personalize** function in the **Saved Search** dropdown list of the Advanced Search dialog.

Figure 2–11 shows the Personalize Saved Search dialog for the **Find Products By Name** search, with **Show in Search List** enabled so that it will appear in the **Saved Search** dropdown list. Note that because this search is not a runtime-defined search, the personalization options appear disabled.

Figure 2–11 Home Page - Advanced Search Dialog - Personalization Option

Where to Find Implementation Details

Following are the sections of the *Oracle Fusion Middleware Fusion Developer's Guide for Oracle Application Development Framework* that describe how to define queries and create query search forms:

- Defining the query for the search form to display

A query is associated with an ADF Business Components view object that you create for the data model project to define a particular query against the database. In particular, a query component is the visual representation of the view criteria defined on that view object. If there are multiple view criteria defined, each of the view criteria can be selected from the **Saved Search** dropdown list. These saved searches are created at design time by the developer. For example, in the Fusion Order Demo application, there are two view criteria defined on the **ProductsVO** view object. When the query associated with that view criteria is run, both view criteria are available for selection. For more information, see [Section 25.1, "Introduction to Creating Search Forms"](#).

- Creating a quick search form

A quick query search form has one search criteria field with a dropdown list of the available searchable attributes from the associated data collection. By default, the searchable attributes are all the attributes in the associated view object. You can exclude attributes by setting the attribute's **Display** control hint to **Hide** in the view object. The user can search against the selected attribute or search against all the displayed attributes. The search criteria field type will automatically match the type of its corresponding attribute type. For more information, see [Section 25.1.2, "Quick Query Search Forms"](#).

- Creating a search form

You create a query search form by dropping a named view criteria item from the Data Controls panel onto a page. You have a choice of dropping only a search panel, dropping a search panel with a results table, or dropping a search panel with a tree table. For more information, see [Section 25.2, "Creating Query Search Forms"](#).

- Displaying the results of a query search

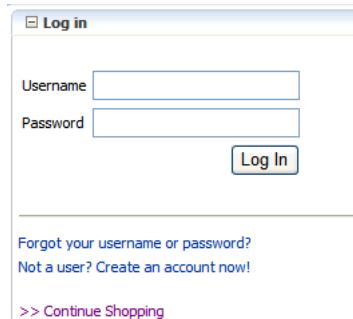
Normally, you would drop a query search panel with the results table or tree table. JDeveloper will automatically wire up the results table or tree table with the query panel. If you drop a query panel by itself and want a results component or if you already have an existing component for displaying the results, you will need to match the query panel's **ResultsComponentId** with the results component's ID. For more information, see [Section 25.2.2, "How to Create a Query Search Form and Add a Results Component Later"](#).

2.3.2 The Login Process

Until you attempt to access secure resources in the storefront site, you are free to browse the product catalog and update the shopping cart anonymously. However, when you click the **My Orders** or **Checkout** links that appear at the top of the home page, you will be challenged by the web container running the site to supply login credentials. The site requires that you enter a valid user name and password before it completes your request to display the linked page.

[Figure 2–12](#) shows the login page fragment that displays before you can view order details or purchase items from the store. For demonstration purposes, enter ngreenbe and welcome1 for the **Username** and **Password**, respectively.

Figure 2–12 Login Region



When you click the **Log In** button, the web container will compare your entries with the credential information stored in its identity store. If the web container is able to authenticate you (because you have entered the user name and password for a registered user), then the web container redirects to the web page specified by your link selection; otherwise, the site prompts you to create an account or to continue browsing as an unauthenticated user.

Where to Find Implementation Details

Following are the sections of the *Oracle Fusion Middleware Fusion Developer's Guide for Oracle Application Development Framework* that describe how to secure Oracle ADF resources so that users are required to log in to access those resources:

- Enabling fine-grained security to secure Oracle ADF resources

Oracle ADF Security is a framework that provides a security implementation that is based on Java Authentication and Authorization Service (JAAS). The Oracle ADF implementation of JAAS is role-based. In JDeveloper, you define these roles and then make permission grants based on these roles to enable fine-grained security for Oracle ADF resources. JDeveloper supports declaratively defining the Oracle ADF policy store for the web pages of an ADF task flow and individual web pages associated with an ADF page definition. For information about securing Oracle ADF resources, see [Section 28.4, "Defining ADF Security Access Policies"](#).

- Triggering dynamic user authentication

When you use Oracle ADF Security, authentication is triggered automatically if the user is not yet authenticated and they try to access a page that is not granted to the `anonymous-role` role. After successfully logging in, another check will be done to verify if the authenticated user has view access to the requested page. For more information, see [Section 28.5, "Handling User Authentication in a Fusion Web Application"](#).

- Performing permission checking within the web page

At runtime, the security policy you define for ADF resources is enforced using standard JAAS permission authorization to determine the user's access rights. If your application requires it, you can use Expression Language (EL) to surface server-side security enforcement directly on the user interface. You might use an EL expression to perform runtime permission checks within the web page to hide components that should not be visible to the user. You can also use Oracle ADF Security's implementation of expressions that are specific to the ADF task flow and ADF page definition to perform permission checks to prevent the user from being able to access a task flow or web page, see [Section 28.6, "Performing Authorization Checks in a Fusion Web Application"](#).

2.3.3 The Ordering Process

You begin the order process by browsing the product catalog. When you click **Add** next to a product, the site updates the shopping cart region to display the item.

[Figure 2–13](#) shows the cart summary with a single item added. The summary displays a subtotal purchase total for the items that appear in the cart.

Figure 2–13 Home Page - Shopping Cart Summary



When you are satisfied with the items in the cart, you can complete the order by clicking the **Checkout** link at the top of the home page. To check out and complete the order, you must become an authenticated user, as described in [Section 2.3.2, "The Login Process"](#).

After you log in, the site displays the checkout page with order details, such as the name and address of the user you registered as. The order is identified by an **Order Information** number that is generated at runtime and assigned to the order. An Order

Summary region displays the order items that comprise the new order. This region is similar to the cart summary on the home page, except that it adds the cost of shipping and deducts any discounts that apply to the order to calculate the total purchase amount.

[Figure 2–14](#) shows the checkout page with an order comprising four order items.

Figure 2–14 Checkout Page - Order Details Form

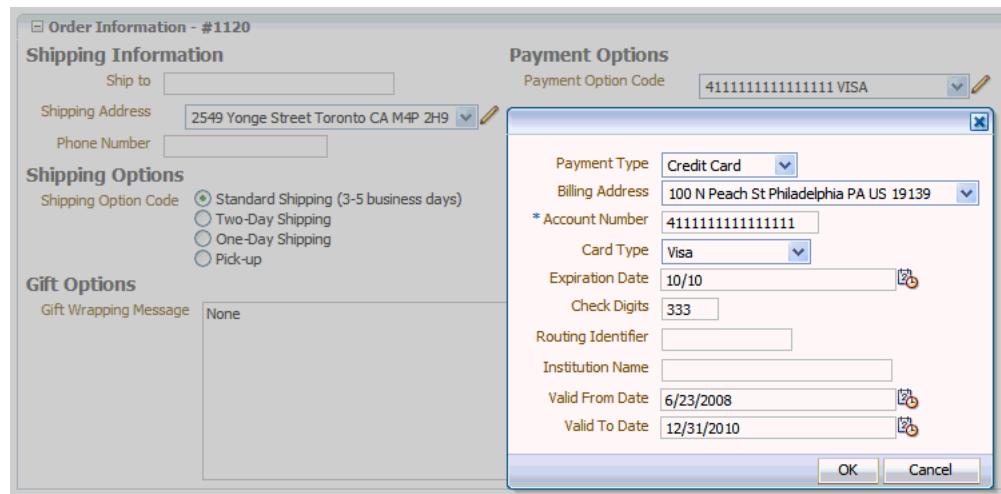
Order Summary	
Zune 30Gb	Audio and Video 225.99 -quantity: 1
Nintendo DS	Games 129.99 -quantity: 1
Tungsten E PDA	Audio and Video 195.99 -quantity: 1
PlayStation 2 Video Game	Games 199.95 -quantity: 1

Items: >751.92
Shipping & Handling >39.09
Discounts >20.00
Your Total: >771.01

Submit Order

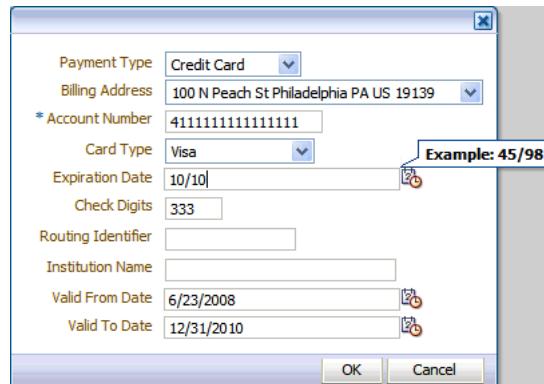
You can use the checkout page to customize details of the order information. For example, click the **Edit** icon next to the **Payment Option Code** field to display and edit payment funding information for the order.

[Figure 2–15](#) shows the detail dialog for the **Payment Option Code** field.

Figure 2–15 Checkout Page - Payment Option Detail Dialog

Many of the fields of the payment options dialog offer user interface hints that guide you to enter specific information.

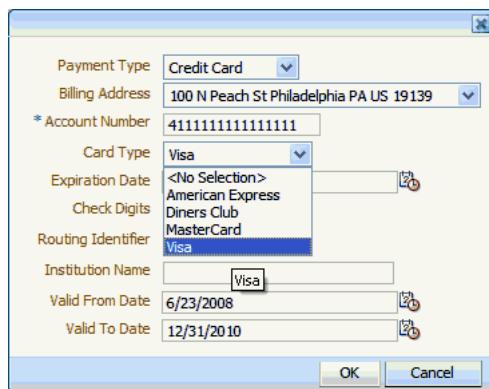
[Figure 2–16](#) shows an example of a date entry (45/98) that the format mask (mm/yy) defines for the **Expiration Date** field.

Figure 2–16 Checkout Page - Payment Options Detail Dialog - Date Format Mask

The **Card Type** field displays a dropdown list that allows you to select from a valid list of values.

[Figure 2–17](#) displays the list of values for the **Card Type** field.

Figure 2–17 Checkout Page - Payment Options Detail Dialog - List of Values (LOV) Choice List



If you close the payment options dialog and click the **Submit Order** button in the checkout page, the purchase order would normally be created and sent into a process flow. However, for this version of the Fusion Order Demo application, the order is simply committed to the database.

Where to Find Implementation Details

Following are the sections of the *Oracle Fusion Middleware Fusion Developer's Guide for Oracle Application Development Framework* that describe how to develop forms like the one used in the order checkout process:

- Creating a databound input form

When you want to create an input form that collects values from the user, instead of having to drop individual attributes, JDeveloper allows you to drop all attributes for an object at once as an input form. The actual UI components that make up the form depend on the type of form dropped. You can create forms that display values, forms that allow users to edit values, and forms that collect values. For more information, see [Section 20.6, "Creating an Input Form"](#).

- Defining format masks for input forms

Oracle ADF Business Components supports the ability to define control hints on attributes of view objects and entity objects in the data model project. Control hints are additional attribute settings that the view layer can use to automatically display the queried information to the user in a consistent, locale-sensitive way. The format mask is one of the hints that you can define. JDeveloper stores the hints in resource bundle files that you can easily localize for multilingual applications. For more information about specifying user interface hints in a data model project, see [Section 5.12, "Defining Attribute Control Hints for View Objects"](#).

- Defining a list of values for input fields

Input forms displayed in the user interface can utilize attributes that you define in the data model project to predetermine a list of values (LOV) for individual fields. When the user submits the form with their selected values, Oracle ADF data bindings in the Oracle ADF Model layer update the value on the view object attributes corresponding to the databound fields. To facilitate this common design task, Oracle ADF Business Components provides declarative support to specify the LOV usage in the user interface. For more information about defining LOV-enabled attributes in a data model project, see [Section 5.11, "Working with List of Values \(LOV\) in View Object Attributes"](#).

- Keeping track of transient session information

When you create a data model project that maps attributes to columns in an underlying table, your ADF entity objects can include transient attributes that display calculated values (for example, using Java or Groovy expressions) or that are value holders. For example, you might define an `Orders` entity object with a transient attribute, such as `InvoiceTotal`, to calculate the sum of `CalculatedOrderTotal` and `TotalShippingCost` attributes. In the Fusion Order Demo application, the order summary region displays the value of the `InvoiceTotal` attribute. Because the attribute is used in the user interface, the actual expression is defined on the view object attribute that is used to specify the databound region. The entity object that the view object references, defines the attribute itself. For more information about defining transient attributes in the data model project, see [Section 4.13, "Adding Transient and Calculated Attributes to an Entity Object"](#).

2.3.4 The Customer Registration Process

The site requires that you become an authenticated user before you can display the checkout page. To make it possible for new customers complete the order process, the site needs to provide a way to guide users through customer registration. To begin, click the registration link on the home page and then click **Create a new customer**.

Customer registration progresses in steps, with one screen dedicated to each step. To represent the progression of these steps, the registration page displays a series of train buttons labelled **Basic Information**, **Address**, **Payment Options**, and **Review**. To navigate the customer registration process, you can click certain train buttons or you can click the **Next** button.

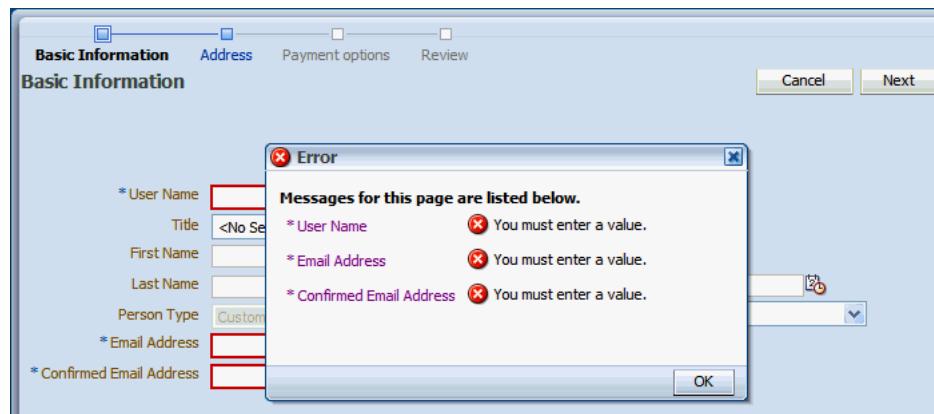
[Figure 2–18](#) shows the first screen in the customer registration process. The train button bar displays the **Basic Information** button as enabled and highlighted to identify the current stop. Notice that the next train button, **Address**, is enabled but not highlighted, while the **Payment Options** and **Review** train buttons appear disabled and grayed out. Together, these train buttons signify that you must complete the activity in a sequential flow.

Figure 2–18 Customer Registration Page - Basic Information Form

The screenshot shows the 'Basic Information' step of a customer registration process. At the top, there are tabs for 'Basic Information', 'Address', 'Payment options', and 'Review'. The 'Basic Information' tab is selected. On the right, there are 'Cancel' and 'Next' buttons. The main area contains several input fields: 'User Name' (marked with an asterisk), 'Title' (dropdown), 'First Name', 'Last Name', 'Person Type' (dropdown, set to 'Customer'), 'Email Address' (marked with an asterisk), 'Confirmed Email Address' (marked with an asterisk), 'Phone', 'Mobile Phone', 'Gender' (dropdown), 'Date of Birth' (calendar icon), 'Contact Method' (dropdown), 'Marital Status' (dropdown), and 'Income'. Below these fields are two sections: 'Categories of interest' (list of items like Audio and Video, Books, Camera and Photo, etc.) and 'I am interested in...' (list box). Between them is a 'Move' button with arrows for moving items between the lists.

Before you enter any information into the Basic Information form, click the **Address** train button. The page displays an error dialog to inform you that specific fields require a value before you can progress to the next step.

Figure 2–19 shows the error dialog with messages stating that the Basic Information form requires a user name and an email address.

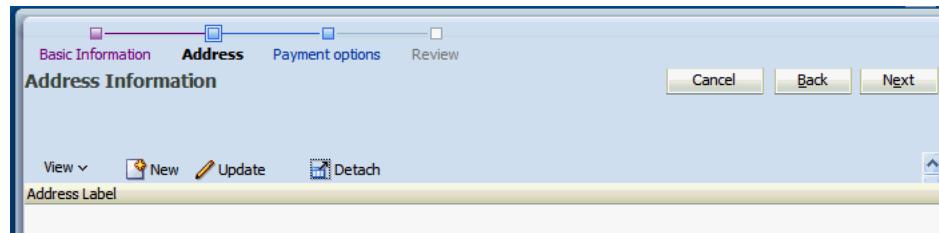
Figure 2–19 Customer Registration Page - Basic Information Form with Validation Error Popup

Click **OK** to dismiss the error dialog. Then enter a user name and email address. Be sure to confirm the email address in the form.

Again, click **Next** to progress to the next task. This time, the site should display the Address screen with icon buttons that you can select to create a new address record in the database (or, in the case of an existing customer, to update an existing address record).

[Figure 2–20](#) shows the Address screen with one column for **Address Label** and no row information. Because you are entering information as a new customer, no address record currently exists, so no rows are available to display below these columns.

Figure 2–20 Customer Registration Page - Address Input Task



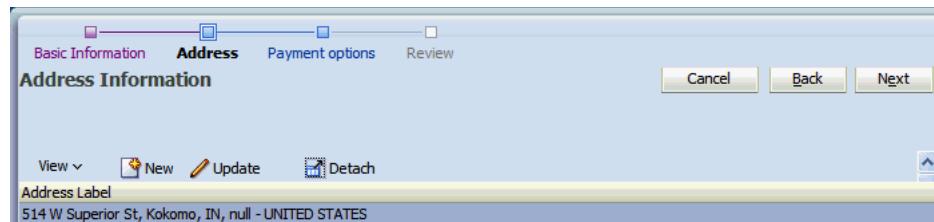
Click **New**. The registration page changes to display an address input form and the current train stop remains on **Address**.

[Figure 2–21](#) shows the empty address input form.

Figure 2–21 Customer Registration Page - Address Input Form

For the three fields with an asterisk symbol (*), enter the address information specified. The asterisk symbol indicates that the value is required. Note that you must also select a country from the dropdown list since this information is required by the database. Then click **Save & Return** to create the new address record in the database.

[Figure 2–22](#) shows the Address screen with the row information for the new address record.

Figure 2–22 Customer Registration Page - Address Record Complete

This concludes the tour of the Fusion Order Demo application.

Where to Find Implementation Details

Following are the sections of the *Oracle Fusion Middleware Fusion Developer's Guide for Oracle Application Development Framework* that describe how to create a task flow like the one used in the registration process:

- Grouping activities using a bounded task flow

A bounded task flow is a specialized form of task flow that 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. An ADF bounded task flow allows reuse, parameters, transaction management, and reentry. Every Fusion web application contains an unbounded task flow, which contains the entry points to the application. The application can then call bounded task flows from activities within the unbounded task flow. For more information, see [Section 13.1, "Introduction to ADF Task Flows"](#) and see [Section 17.6.1, "ADF Bounded Task Flows as Trains"](#).

- Displaying the task flow using a train component

You can create a train based on activities that you define in an ADF bounded task flow that specifies the `<train/>` element in its metadata. A bounded task flow is a specialized form of task flow that 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. An ADF bounded task flow allows reuse, parameters, transaction management, and reentry. For more information, see [Section 17.6.3, "How to Create a Train"](#).

- Requiring values to complete an input form

The input form displays attributes of a data collection you drop from the Data Controls panel. By default, the component sets the `mandatory` control hint for these attributes to `false`, which means that the `required` attribute on the component will evaluate to `false` as well. You can override this control hint by setting the `required` attribute on the component to `true`. If you decide that all instances of the attribute should be mandatory, you can change the `mandatory` property on the attribute where it is defined by the ADF Business Components entity object. The entity object is a data model component that represents a row from a specific table in the database and that simplifies modifying its associated attributes. For more information, see [Section 20.2, "Using Attributes to Create Text Fields"](#) and see [Section 4.10, "Setting Attribute Properties"](#).

Part II

Building Your Business Services

Part II contains the following chapters:

- [Chapter 3, "Getting Started with ADF Business Components"](#)
- [Chapter 4, "Creating a Business Domain Layer Using Entity Objects"](#)
- [Chapter 5, "Defining SQL Queries Using View Objects"](#)
- [Chapter 6, "Working with View Object Query Results"](#)
- [Chapter 7, "Defining Validation and Business Rules Declaratively"](#)
- [Chapter 8, "Implementing Validation and Business Rules Programmatically"](#)
- [Chapter 9, "Implementing Business Services with Application Modules"](#)
- [Chapter 10, "Sharing Application Module View Instances"](#)
- [Chapter 11, "Using Oracle ADF Model in a Fusion Web Application"](#)
- [Chapter 12, "Integrating Web Services Into a Fusion Web Application"](#)

Getting Started with ADF Business Components

This chapter provides an overview of the ADF Business Components layer of Oracle ADF, including a description of the key features they provide for building your business services.

This chapter includes the following sections:

- [Section 3.1, "Introduction to ADF Business Components"](#)
- [Section 3.2, "Comparison to Familiar 4GL Tools"](#)
- [Section 3.3, "Overview of Design Time Facilities"](#)
- [Section 3.4, "Overview of the UI-Aware Data Model"](#)
- [Section 3.5, "Overview of the Implementation Architecture"](#)
- [Section 3.6, "Overview of Groovy Support"](#)

3.1 Introduction to ADF Business Components

ADF Business Components and JDeveloper simplify the development, delivery, and customization of business applications for the Java EE platform. With ADF Business Components, developers aren't required to write the application infrastructure code required by the typical Java EE application to:

- Connect to the database
- Retrieve data
- Lock database records
- Manage transactions

ADF Business Components addresses these tasks through its library of reusable software components and through the supporting design time facilities in JDeveloper. Most importantly, developers save time using ADF Business Components since the JDeveloper design time makes typical development tasks entirely declarative. In particular, JDeveloper supports declarative development with ADF Business Components to:

- Author and test business logic in components which automatically integrate with databases
- Reuse business logic through multiple SQL-based views of data, supporting different application tasks

- Access and update the views from browser, desktop, mobile, and web service clients

The goal of ADF Business Components is to make the business services developer more productive.

3.1.1 ADF Business Components Features

ADF Business Components provides a foundation of Java classes that allow your business-tier application components to leverage the functionality provided in the following areas:

Simplifying Data Access

- Design a data model for client displays, including only necessary data
- Include master-detail hierarchies of any complexity as part of the data model
- Implement end-user Query-by-Example data filtering without code
- Automatically coordinate data model changes with business domain object layer
- Automatically validate and save any changes to the database

Enforcing Business Domain Validation and Business Logic

- Declaratively enforce required fields, primary key uniqueness, data precision-scale, and foreign key references
- Easily capture and enforce both simple and complex business rules, programmatically or declaratively, with multilevel validation support
- Navigate relationships between business domain objects and enforce constraints related to compound components

Supporting Sophisticated UIs with Multipage Units of Work

- Automatically reflect changes made by business service application logic in the user interface
- Retrieve reference information from related tables, and automatically maintain the information when the user changes foreign-key values
- Simplify multistep web-based business transactions with automatic web-tier state management
- Handle images, video, sound, and documents without having to use code
- Synchronize pending data changes across multiple views of data
- Consistently apply prompts, tooltips, format masks, and error messages in any application
- Define custom metadata for any business components to support metadata-driven user interface or application functionality
- Add dynamic attributes at runtime to simplify per-row state management

Implementing High-Performance Service-Oriented Architecture

- Support highly functional web service interfaces for business integration without writing code
- Enforce best-practice interface-based programming style
- Simplify application security with automatic JAAS integration and audit maintenance

- "Write once, run anywhere": use the same business service as plain Java class, EJB session bean, or web service

3.1.2 ADF Business Components Core Objects

ADF Business Components implements the business service through the following set of cooperating components:

- Entity object

An *entity object* represents a row in a database table and simplifies modifying its data by handling all data manipulation language (DML) operations for you. It can encapsulate business logic for the row to ensure that your business rules are consistently enforced. You associate an entity object with others to reflect relationships in the underlying database schema to create a layer of business domain objects to reuse in multiple applications.

- View object

An *view object* represents a SQL query. You use the full power of the familiar SQL language to join, filter, sort, and aggregate data into exactly the shape required by the end-user task. This includes the ability to link a view object with others to create master-detail hierarchies of any complexity. When end users modify data in the user interface, your view objects collaborate with entity objects to consistently validate and save the changes.

- Application module

An *application module* is the transactional component that UI clients use to work with application data. It defines an updatable data model and top-level procedures and functions (called *service methods*) related to a logical unit of work related to an end-user task.

While the base components handle all the common cases through built-in behavior, customization is always possible and the default behavior provided by the base components can be easily overridden or augmented.

3.2 Comparison to Familiar 4GL Tools

ADF Business Components provides components that implement functionality similar to that offered by enterprise 4GL tools. Several key components in ADF Business Components map to concepts that you may be familiar with in other 4GL tools.

3.2.1 Familiar Concepts for Oracle Forms Developers

ADF Business Components implements all of the data-centric aspects of the familiar Oracle Forms runtime functionality, but in a way that is independent of the user interface. In Oracle Forms, each *form* contains both visual objects (like canvases, windows, alerts, and LOVs), as well as nonvisual objects (like data blocks, relations, and record groups). Individual data block *items* have both visual properties like Foreground Color and Bevel, as well as nonvisual properties like Data Type and Maximum Length. Even the different event-handling triggers that Forms defines fall into visual and nonvisual categories. For example, it's clear that triggers like WHEN-BUTTON-PRESSED and WHEN-MOUSE-CLICKED are visual in nature, relating to the front-end UI, while triggers like WHEN-VALIDATE-ITEM and ON-INSERT are more related to the backend data processing. While merging visual and nonvisual aspects definitely simplifies the learning curve, the flip side is that it can complicate reuse. With a cleaner separation of UI-related and data-related elements, it would be

easier to redesign the user interface without disturbing backend business logic and easier to repurpose back-end business logic in multiple different forms.

In order to imagine this separation of UI and data, consider reducing a form as you know it to only its nonvisual, data-related aspects. This reduces the form to a container of data blocks, relations, and record groups. This container would continue to provide a database connection for the data blocks to share and would be responsible for coordinating transaction commits or rollbacks. Of course, you could still use the *nonvisual* validation and transactional triggers to augment or change the default data-processing behavior as well. This nonvisual object you are considering is a kind of a "smart data model" or a generic application module, with data and business logic, but no user interface elements. The goal of separating this application module from anything visual is to allow any kind of user interface you need in the future to use it as a data service.

Focus a moment on the role the data blocks would play in this application module. They would query rows of data from the database using SQL, coordinate master/detail relationships with other data blocks, validate user data entry with WHEN-VALIDATE-RECORD and WHEN-VALIDATE-ITEM triggers, and communicate valid user changes back to the database with INSERT, UPDATE, and DELETE statements when you commit the data service's transaction.

Experience tells you that you need to filter, join, order, and group data for your end-users in a variety of ways to suit the many different tasks. On the other hand, the validation rules that you apply to your business domain data remain basically the same over time. Given these observations, it would be genuinely useful to write business entity validation exactly once, and leverage it consistently anywhere that data is manipulated by users in your applications.

Enabling this flexibility requires further "factoring" of your data block functionality. You need one kind of "SQL query" object to represent each of the many different views of data your application requires, and you need another kind of "business entity" object to enforce business rules and communicate changes to your base table in a consistent way. By splitting things like this, you can have multiple "view objects" with specific SQL queries that present the same business data yet each working with the same underlying "entity object."

Oracle ADF addresses the UI/data split above by providing ready-to-use Java components that implement typical Forms functionality. Responsibilities between the querying and entity-related functions are cleanly separated, resulting in better reuse.

3.2.1.1 Similarities Between the Application Module and a "Headless" Form Module

The application module component is the "data portion" of the form. The application module is a smart data service containing a data model of master-detail-related queries that your client interface needs to work with. It also provides a transaction and database connection used by the components it contains. It can contain form-level procedures and functions, referred to as service methods, that are encapsulated within the service implementation. You can decide which of these procedures and functions should be private and which ones should be public.

3.2.1.2 Similarities Between the Entity Object and a Forms Record Manager

The entity object component implements the "validation and database changes" portion of the data block functionality. In the Forms runtime, this duty is performed by the record manager. The record manager is responsible for keeping track of which of the rows in the data block have changed, for firing the block-level and item-level validation triggers when appropriate, and for coordinating the saving of changes to the database. This is exactly what an entity object does for you. The entity object is a

component that represents your business domain entity through an underlying database table. The entity object gives you a single place to encapsulate business logic related to validation, defaulting, and database modification behavior for that business object.

3.2.1.3 Similarities Between the View Object and a Data Block

The `ViewObject` component performs the "data retrieval" portion of the data block functionality. Each view object encapsulates a SQL query, and at runtime each one manages its own query result set. If you connect two or more view objects in master-detail relationships, that coordination is handled automatically. While defining a view object, you can link any of its query columns to underlying entity objects. By capturing this information, the view object and entity object can cooperate automatically for you at runtime to enforce your domain business logic, regardless of the "shape" of the business data required by the user's task.

3.2.2 Familiar Concepts for PeopleTools Developers

If you have developed solutions in the past with PeopleTools, you are familiar with the PeopleTools component structure. ADF Business Components implement the data access functionality you are familiar with from PeopleTools.

3.2.2.1 Similarities Between the Application Module and a "Headless" Component

Oracle ADF adheres to an MVC pattern and separates the model from the view. Pages, which you are familiar with in the PeopleTools Component, are defined in the view layer, using standard technologies like JSF and ADF Faces components for web-based applications or Swing for desktop-fidelity client displays.

The ADF application module defines the data structure, just like the PeopleTools Component Buffer does. By defining master-detail relationships between ADF query components that produce row sets of data, you ensure that any application module that works with the data can reuse the natural hierarchy as required, similar to the scroll levels in the Component Buffer.

Similar to the Component Interface you are familiar with, the application module is a service object that provides access to standard methods, as well as additional developer-defined business logic. In order to present a "headless" data service for a particular user interface, the Component Interface restricts a number of PeopleTools functions that are related to UI interaction. The application module is similar to the Component Interface in that it provides a "headless" data service, but in contrast it does not do this by wrapping a restricted view of an existing user interface. Instead, the application module is architected to deal exclusively with business logic and data access. Rather than building a Component Interface on top of the component, with ADF Business Components you first build the application module service that is independent of user interface, and then build one or more pages on top of this service to accomplish some end-user task in your application.

The application module is associated with a transaction object in the same way that the PeopleTools Component Buffer is. The application module also provides a database connection for the components it contains. Any logic you associate today with the transaction as Component PeopleCode, in ADF Business Components you would define as logic on the application module.

Logic associated with records in the transaction, that today you write as Component Record PeopleCode or Component Record Field PeopleCode, should probably *not* be defined on the application module. ADF Business Components has view objects that allow for better re-use when the same record appears in different components.

In summary, PeopleTools uses the component for the container concept, whereas ADF Business Components uses the application module. That is where the similarity ends. Do not assume that all of your component code will migrate to an application module. First, understand the concept of the view object, which is the layer between the entity object and the application module. Then, decide which of your component code is suitable for an application module and which is suitable for view objects.

3.2.2.2 Similarities Between the Entity Object and a Record Definition

The entity object is the mapping to the underlying data structure, just like the PeopleTools Record Definition maps to the underlying table or view. You'll often create one entity object for each of the tables that you need to manipulate your application.

Similar to how you declare a set of valid values for fields like "Customer Status" using PeopleTools' translate values, in ADF Business Components you can add declarative validations to the individual attributes of an entity object. Any logic you associate with the record that applies throughout your applications, which today you write as Record PeopleCode or Record Field PeopleCode, can be defined in ADF Business Components on the entity object.

3.2.2.3 Similarities Between the View Object and a Row Set

Just like a PeopleTools row set, a view object can be populated by a SQL query. Unlike a row set, a view object definition can contain business logic.

Any logic which you would find in Component Record PeopleCode is a likely candidate to define on the view object. Component Record PeopleCode is directly tied to the component, but a view object can be associated with different application modules. Whereas you can use the same record definition in many PeopleTools components, Oracle ADF allows you to reuse the business logic across multiple applications.

The view object queries data in exactly the "shape" that is useful for the current application. Many view objects can be built on top of the same entity object.

You can define relationships between view objects to create master-detail structures, just as you find them in the scroll levels in the PeopleTools component.

3.2.3 Familiar Concepts for Siebel Tools Developers

If you have developed solutions in the past with Siebel Tools version 7.0 or earlier, you will find that ADF Business Components implements all of the familiar data access functionality you are familiar with, with numerous enhancements.

3.2.3.1 Similarities Between the entity Object and a Table Object

Like the Siebel Table object, the ADF entity object describes the physical characteristics of a single table, including column names and physical data types. Both objects contain sufficient information to generate the DDL (data definition language) statements to create the physical tables in the database. In ADF Business Components you define associations between entity objects to reflect the foreign keys present in the underlying tables. These associations allow view object queries used by user interface pages to automatically join business information. ADF Business Components handles list of values (LOV) objects that you reference from data columns through a combination of declarative entity-level validation rules and view object attribute-level LOV definitions. You can also encapsulate other declarative or programmatic business logic with these entity object "table" handlers that is automatically reused in any view of the data you create.

3.2.3.2 Similarities Between the View Object and a Business Component

Like the Siebel Business Component, the ADF view object describes a logical mapping on top of the underlying physical table representation. Both the Siebel Business Component and the ADF view object allow you to provide logical field names, data, and calculated fields that match the needs of the user interface. As with the Siebel Business Component, with the ADF view object you can define view objects that join information from various underlying tables. The related ADF view link is similar to the Siebel Link object and allows you to define master-detail relationships. In ADF Business Components, your view object definitions can exploit the full power of the SQL language to shape the data as required by the user interface.

3.2.3.3 Similarities Between the Application Module and a Business Object

The Siebel Business Object lets you define a collection of business components. The ADF application module performs a similar task, allowing you to create a collection of master-detail view objects that act as a "data model" for a set of related user interface pages. In addition, the application module provides a transaction and database connection context for this group of data views. You can make multiple requests to objects obtained from the application module and these participate in the same transaction.

3.2.4 Familiar Functionality for ADO.NET Developers

If you have developed solutions in the past with Visual Studio 2003 or 2005, you are familiar with using the ADO.NET framework for data access. ADF Business Components implements all of the data access functionality you are familiar with from ADO.NET, with numerous enhancements.

3.2.4.1 Similarities Between the Application Module and a Data Set

The application module component plays the same role as the ADO.NET data set. It is a strongly typed service component that represents a collection of row sets called view object instances, which are similar to ADO.NET data tables. An application module exposes a service interface that surfaces the rows of data in a developer-configurable set of its view instances as an SDO-compatible service (accessible as a web service, or as an SCA composite). The application module works with a related transaction object to provide the context for the SQL queries that the view objects execute. The application module also provides the context for modifications saved to the database by the entity objects, which play the role of the ADO.NET data adapter.

3.2.4.2 Similarities Between the Entity Object and a Data Adapter

The entity object component is like a strongly-typed ADO.NET data adapter. It represents the rows in a particular table and handles the find-by-primary-key, insert, update, delete, and lock operations for those rows. In ADF Business Components, you don't have to specify these statements yourself, but you can override them if you need to. The entity object encapsulates validation or other business logic related to attributes or entire rows in the underlying table. This validation is enforced when data is modified and saved by the end user using any view object query that references the underlying entity object. One difference in ADF Business Components is that the arbitrary, flexible querying is performed by SQL statements at the view object instance level, but the view objects and entity objects coordinate automatically at runtime.

3.2.4.3 Similarities Between the View Object and a Data Table

The view object component encapsulates a SQL query and manages the set of resulting rows. It can be related to an underlying entity object to automatically coordinate

validation and saving of modifications made by the user to those rows. This cooperation between a view object's queried data and an entity object's encapsulated business logic offers all of the benefits of the data table with the clean encapsulation of business logic into a layer of business domain objects. Like ADO.NET data tables, you can easily work with a view object's data as XML or have a view object read XML data to automatically insert, update, or delete rows based on the information it contains.

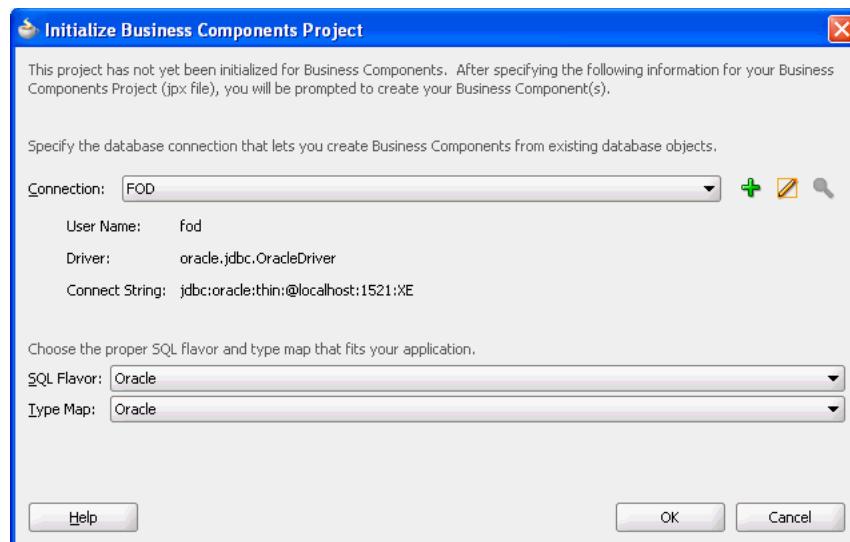
3.3 Overview of Design Time Facilities

JDeveloper includes comprehensive design time support for ADF Business Components. Collectively, these facilities let you create, edit, diagram, test, and refactor the business components.

3.3.1 Choosing a Connection, SQL Flavor, and Type Map

The first time you create a component, you'll see the Initialize Business Components Project dialog shown in [Figure 3–1](#). You use this dialog to select a design time application resource connection to use while working on your business components in this data model project or to create a new application resource connection by copying an existing IDE-level connection.

Figure 3–1 Initialize Business Components Project Dialog



Since this dialog appears before you create your first business component, you also use it to globally control the SQL flavor that the view objects will use to formulate SQL statements. Although the default for an Oracle database connection is always the Oracle SQL flavor, other SQL flavors you can choose include OLite (for the Oracle Lite database), SQLServer for a Microsoft SQLServer database, DB2 for an IBM DB2 database, and SQL92 for any other supported SQL92-compliant database.

The dialog also lets you determine which set of data types that you want the data model project to use. If JDeveloper detects you are using an Oracle database driver, it defaults the **Type Map** setting to the Oracle type map and will use the optimized types in the `oracle.jbo.domain` package. You can change this setting to globally use only the basic Java data types.

Note: If you plan to have your application run against both Oracle and non-Oracle databases, you should select the **SQL92** SQL flavor when you begin building your application, not later. While this sacrifices using some of the Oracle-specific optimizations that are inherent in using the Oracle SQL flavor, it makes the application portable to both Oracle and non-Oracle databases.

3.3.2 Creating New Components Using Wizards

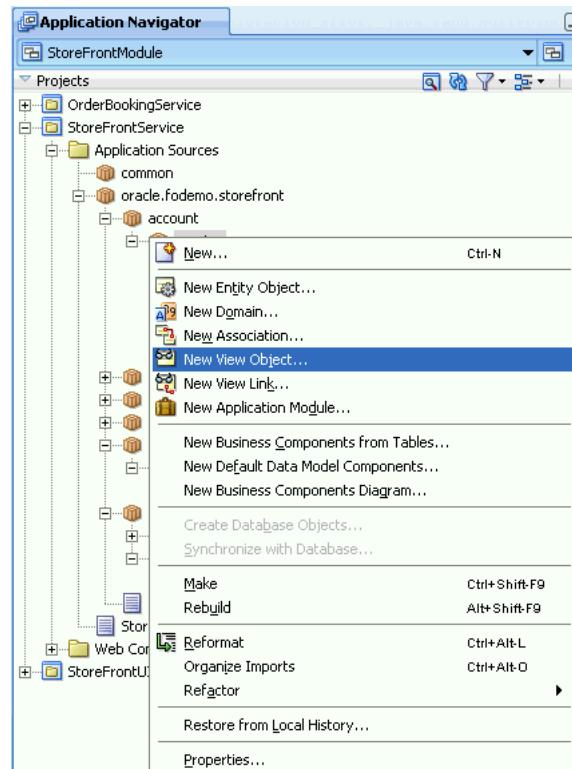
In the New Gallery in the **ADF Business Components** category, JDeveloper offers a wizard to create each kind of business component. Each wizard allows you to specify the component name for the new component and to select the package into which you'd like to organize the component. If the package does not yet exist, the new component becomes the first component in that new package.

The wizard presents a series of panels that capture the necessary information to create the component type. When you click **Finish**, JDeveloper creates the new component by saving its XML component definition file. If you have set your Java generation options to generate classes by default, JDeveloper also creates the initial custom Java class files.

3.3.3 Creating New Components Using the Context Menu

Once a package exists in the Application Navigator, you can quickly create additional business components of any type in the package by selecting it in the Application Navigator and using one of the options on the context menu shown in [Figure 3–2](#).

Figure 3–2 Context Menu Options on a Package to Create Any Kind of Business Component



3.3.4 Editing Components Using the Component Overview Editor

Once a component exists, you can edit it using the respective overview editor that you access either by double-clicking on the component in the Application Navigator or by selecting it and choosing the **Edit** option from the context menu. The overview editor presents the same editing options that you see in the wizard but it may arrange them differently. The overview editor allows you to change any aspect of the component. When you click **OK**, JDeveloper updates the components XML component definition file and, if necessary, any of its related custom Java files. Because the overview editor is a JDeveloper editor window, rather than a modal dialog, you can open and view the overview editor for as many components as you require.

3.3.5 Visualizing, Creating, and Editing Components Using UML Diagrams

JDeveloper offers extensive UML diagramming support for ADF Business Components. You can drop components that you've already created onto a business components diagram to visualize them. You can also use the diagram to create and modify components. The diagrams are kept in sync with changes you make in the editors.

To create a new business components diagram, use the **Business Components Diagram** item in the **ADF Business Components** category of the JDeveloper New Gallery. This category is part of the **Business Tier** choices.

3.3.6 Testing Application Modules Using the Business Component Browser

Once you have created an application module component, you can test it interactively using the built-in Business Component Browser. To launch the Business Component Browser, select the application module in the Application Navigator or in the business components diagram and choose either **Run** or **Debug** from the context menu.

The Business Component Browser presents the view object instances in the application module's data model and allows you to interact with them using a dynamically generated user interface. The tool also provides a list of the application module's client interface methods that you can test interactively by double-clicking the application module node. This tool is invaluable for testing or debugging your business service both before and after you create the web page view layer.

3.3.7 Refactoring Components

At any time, you can select a component in the Application Navigator and choose **Refactor > Rename** from the context menu to rename the component. The Structure window also provides a **Rename** context menu option for details of components, such as view object attributes or view instances of the application module data model, that do not display in the Application Navigator. You can also select one or more components in the navigator by using Ctrl + click and then choosing **Refactor > Move** from the context menu to move the selected components to a new package. References to the old component names or packages in the current data model project are adjusted automatically.

3.4 Overview of the UI-Aware Data Model

One of the key simplifying benefits of using ADF Business Components for your business service implementation is the application module's support for a "UI-aware data model" of row sets. The data model defines the business objects specific to your application, while the row sets of each business object contain the data. In the UI

portion of the application, the UI components interact with these business objects to perform retrieve, create, edit, and delete operations. When you use ADF Business Components in combination with the ADF Model layer and ADF Faces UI components, the data model is "UI aware" because your UI components will automatically update to reflect any changes to the row sets of these business objects.

Thus, the UI-aware data model represents a solution that works across application technology layers to ensure that the UI and data model remain synchronized.

3.4.1 A More Generic Business Service Solution

Using a typical Java EE business service implementation makes the client developer responsible for:

- Invoking service methods to return data to present
- Tracking what data the client has created, deleted, or modified
- Passing the changes back to one or more different service methods to validate and save them

Retrieving, creating, editing, deleting, and saving is a typical sequence of tasks performed during application development. As a result, the ADF application module represents a smarter, more generic solution. Using the application module for your business service, you simply bind client UI components like fields, tables, and trees to the active view object instances in the application module's data model. Your UI components in JSP or JSF pages for the web or mobile devices (as well as desktop-fidelity UIs comprising windows and panels that use Swing) automatically update to reflect any changes to the rows in the view object row sets of the data model. This active data notification also extends to custom business service methods that happen to produce changes to the data model.

Under the covers, the application module component implements a set of *generic* service methods that allow users to leverage its UI-aware data model in a service-oriented architecture (SOA). Both web service and UI clients can easily access an application module's data model using simple APIs. These APIs enable you to search for and modify any information that the application module makes available.

When you build UIs that take advantage of the ADF Model layer for declarative data binding, you generally won't need to write client-side code. Because the data model is UI-aware, your UI components will be bound declaratively to view objects in the data model and to custom business service methods.

3.4.2 Typical Scenarios for a UI-Aware Data Model

Without a UI-aware data model, you would need to write more code in the client to handle the straightforward, everyday CRUD-style operations. In addition, to keep pages up to date, you would need to manage "refresh flags" that clue the controller layer in to requesting a "repull" of data from the business service to reflect data that might have been modified. When using an ADF application module to implement your business service, you can focus on the business logic at hand, instead of the plumbing to make your business work as your end users expect.

Consider the following three simple, concrete examples of the UI-aware data model:

- New data appears in relevant displays without requerying

A customer logs into the StoreFrontModule of Fusion Order Demo application and displays a list of items in their shopping cart. Then if the customer visits some product pages and creates a new order item, when they return back to display

their shopping cart, the new item appears in their list without requiring the application to requery the database.

- Changes caused by business domain logic automatically reflected

A back office application causes an update to the order status. Business logic encapsulated in the Orders entity object in the business domain layer contains a simple rule that updates the last update date whenever the order status attribute is changed. The user interface updates to automatically reflect the last update date that was changed by the logic in the business domain layer.
- Invocation of a business service method requeries data and sets current rows

In a tree display, the user clicks on a specific node in a tree. This action declaratively invokes a business service method on your application module that requeries master-detail information and sets the current rows to an appropriate row in the row set. The display updates to reflect the new master-detail data and current row displayed.

3.4.3 UI-Aware Data Model Support for Custom Code

Because the application module supports the UI-aware data model, your client user interface will remain up to date. This means you will not need to write code in the client that is related to setting up or manipulating the data model.

Another typical type of client-side code you no longer have to write using ADF Business Components is code that coordinates detail data collections when a row in the master changes. By linking the view objects, you can have the coordination performed automatically for you.

However, when you do need to write custom code, encapsulating that code inside custom methods of your application module component is a best practice. For example, whenever the programmatic code that manipulates view objects is a logical aspect of implementing your complete business service functionality, you should encapsulate the details by writing a custom method in your application module's Java class. This includes, but is not limited to, code that:

- Configures view object properties to query the correct data to display
- Iterates over view object rows to return an aggregate calculation
- Performs any kind of multistep procedural logic with one or more view objects

By centralizing these implementation details in your application module, you gain the following benefits:

- You make the intent of your code more clear to clients.
- You allow multiple client pages to easily call the same code if needed.
- You simplify regression-testing of your complete business service functionality.
- You keep the option open to improve your implementation without affecting clients.
- You enable *declarative* invocation of logical business functionality in your pages.

3.5 Overview of the Implementation Architecture

Before you begin implementing specific ADF business components, it is a good idea to have some familiarity with the Oracle ADF business services layer's design and implementation.

3.5.1 Standard Java and XML

As is the case with all Oracle ADF technologies, ADF Business Components is implemented in Java. The working, tested components in the framework provide generic, metadata-driven functionality from a rich layer of robust code. ADF Business Components follows the Java EE community best practice of using cleanly separated XML files to store metadata that you define to configure each component's runtime behavior.

Since ADF Business Components is often used for business critical applications, it's important to understand that the full source for Oracle ADF, including the ADF Business Components layer, is available to supported customers through Oracle Worldwide Support. The full source code for Oracle ADF can be an important tool to assist you in diagnosing problems, as described in [Section 29.5, "Using the ADF Declarative Debugger"](#). Working with the full source code for Oracle ADF also helps you understand how to correctly extend the base framework functionality to suit your needs, as described in [Section 33.3, "Customizing Framework Behavior with Extension Classes"](#).

3.5.2 Application Server or Database Independence

Applications built using ADF Business Components can run on any Java-capable application server, including any Java EE-compliant application server. Because business components are implemented using plain Java classes and XML files, you can use them in any runtime environment where a Java Virtual Machine is present. This means that services built using ADF Business Components are easy to use both inside a Java EE server — known as the "container" of your application at runtime — and outside.

Customers routinely use application modules in such diverse configurations as command-line batch programs, web services, custom servlets, JSP pages, and desktop-fidelity clients built using Swing.

You can also build applications that work with non-Oracle databases, as described in [Section 3.3.1, "Choosing a Connection, SQL Flavor, and Type Map"](#). However, applications that target Oracle databases will find numerous optimizations built into ADF Business Components.

3.5.3 Java EE Design Pattern Support

The ADF Business Components layer implements all of the popular Java EE design patterns that you would normally need to understand, implement, and debug yourself to create a real-world enterprise Java EE application. If it is important to you to cross-reference the names of these design patterns from the Java EE specifications with their ADF Business Components counterparts, you can refer to [Appendix F, "ADF Business Components Java EE Design Pattern Catalog"](#).

3.5.4 Source Code Organization

Since ADF Business Components is implemented in Java, its classes and interfaces are organized into packages. Java packages are identified by dot-separated names that developers use to arrange code into a hierarchical naming structure.

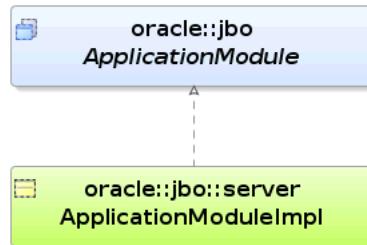
The classes and interfaces that comprise the source code provided by ADF Business Components reside in the `oracle.jbo` package and numerous subpackages. However, in day to day work with ADF Business Components, you'll work typically with classes and interfaces in these two key packages:

- The `oracle.jbo` package, which contains all of the interfaces that are designed for the business service client to work with
- The `oracle.jbo.server` package, which contains the classes that implement these interfaces

Note: The term *client* here refers to any code in the model, view, or controller layers that accesses the application module component as a business service.

Figure 3–3 shows a concrete example of the application module component. The client interface for the application module is the `ApplicationModule` interface in the `oracle.jbo` package. This interface defines the names and signatures of methods that clients can use while working with the application module, but it does not include any specifics about the implementation of that functionality. The class that implements the base functionality of the application module component resides in the `oracle.jbo.server` package and is named `ApplicationModuleImpl`.

Figure 3–3 Oracle ADF Business Components Separate Interface and Implementation



3.5.5 Package Naming Conventions

Since ADF Business Components is implemented in Java, the components of your application (including their classes, interfaces, and metadata files) will also be organized into packages.

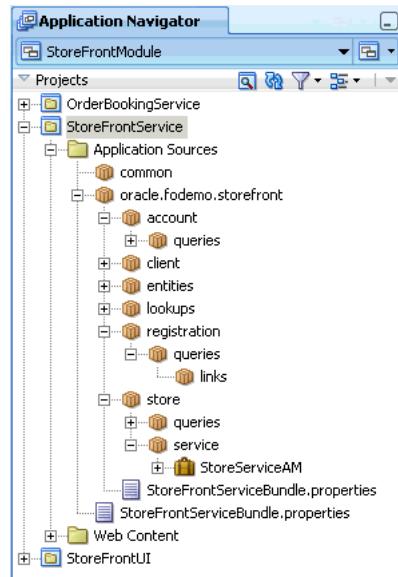
To ensure that your components won't clash with reusable components from other organizations, choose package names that begin with your organization's name or web domain name. So, for example, the Apache organization chose `org.apache.tomcat` for a package name related to its Tomcat web server, while Oracle picked `oracle.xml.parser` as a package name for its XML parser. Components you create for your own applications might reside in packages with names like `com.yourcompany.yourapp` and subpackages of these.

As a specific example, the ADF Business Components that make up the main business service for the Fusion Order Demo application are organized into the `oracle.fodemo.storefront` package and its subpackages. As shown in **Figure 3–4**, these components reside in the `StoreFrontService` project in the `StoreFrontModule` application, and are organized broadly as follows:

- `oracle.fodemo.storefront.store.service` contains the `StoreServiceAM` application module
- `oracle.fodemo.storefront.store.queries` contains the view objects
- `oracle.fodemo.storefront.entities` contains the entity objects

- oracle.fodemo.storefront.design contains UML diagrams documenting the service

Figure 3–4 Organization of ADF Business Components in the Fusion Order Demo Application



In your own applications, you can choose any package organization that you believe best. In particular, keep in mind that you are not constrained to organize components of the same type into a single package.

Because JDeveloper supports component refactoring, you can easily rename components or move them to a different package at any time. This flexibility allows you to easily incorporate inevitable changes into the application as your application evolves.

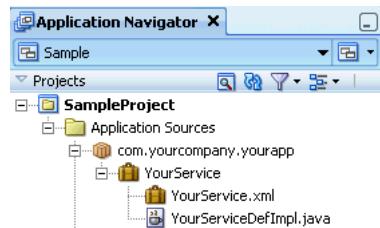
There is no correct number for the optimal number of components in a package. However, with experience, you'll realize that the correct structure for your team falls somewhere between the two extremes of placing all components in a single package and placing each component in its own, separate package.

One thing to consider is that the package in ADF Business Components is the unit of granularity that JDeveloper supports for reuse in other data model projects. So, you might factor this consideration into how you choose to organize components. For more information, see [Section 33.7, "Working with Libraries of Reusable Business Components"](#).

3.5.6 Metadata with Optional Custom Java Code

Each kind of component in ADF Business Components comes with built-in runtime functionality that you control through declarative settings. These settings are stored in an XML component definition file with the same name as the component that it represents. When you need to write custom code for a component, for example to augment the component's behavior, you can enable an optional custom Java class for the component in question. [Figure 3–5](#) shows how the Application Navigator displays the XML component definition and optional custom Java class for an application module.

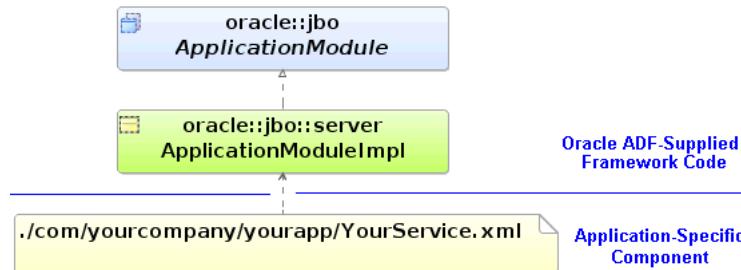
Figure 3–5 Application Navigator Displays Component XML File and Optional Class Files



3.5.6.1 Example of an XML-Only Component

Figure 3–6 illustrates the XML component definition file for an application-specific component like an application module named *YourService* that you create in a package named `com.yourcompany.yourapp`. The corresponding XML component definition resides in a `./com/yourcompany/yourapp` subdirectory of the data model project's source path root directory. That XML file records the name of the Java class it should use at runtime to provide the application module implementation. In this case, the XML records the name of the base `oracle.jbo.server.ApplicationModuleImpl` class provided by Oracle ADF.

Figure 3–6 XML Component Definition File for an Application Module



When used without customization, your component is completely defined by its XML component definition and it will be fully functional without custom Java code or even a Java class file for the component. If you have no need to extend the built-in functionality of a component in ADF Business Components, and no need to write any custom code to handle its built-in events, you can use the component in this XML-only fashion.

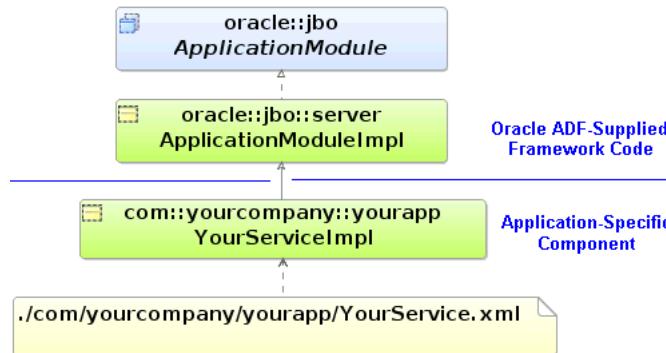
3.5.6.2 Example of a Component with Custom Java Class

When you need to add custom code to extend the base functionality of a component or to handle events, you can enable a custom Java class for any of the key types of ADF Business Components you create. You enable the generation of custom classes for a component on the Java page of its respective overview editor in JDeveloper. When you enable this option, JDeveloper creates a Java source file for a custom class related to the component whose name follows a configurable naming standard. This class, whose name is recorded in the component's XML component definition, provides a place where you can write the custom Java code required by that component. Once you've enabled a custom Java class for a component, you can navigate to it using a corresponding **Go To componentName Class** option in the component's Application Navigator context menu.

Figure 3–7 illustrates what occurs when you enable a custom Java class for the *YourService* application module considered above. A `YourServiceImpl.java`

source code file is created in the same source path directory as your component's XML component definition file. The `YourServiceImpl.xml` file is updated to reflect the fact that at runtime the component should use the `com.yourcompany.yourapp.YourServiceImpl` class instead of the base `ApplicationModuleImpl` class.

Figure 3–7 Component with Custom Java Class



Note: The examples in this guide use default settings for generated names of custom component classes and interfaces. If you want to change these defaults for your own applications, use the Business Components: Class Naming page of the JDeveloper Preferences dialog. Changes you make only affect newly created components.

3.5.7 Basic Data Types

The Java language provides a number of built-in data types for working with strings, dates, numbers, and other data. When working with ADF Business Components, you can use these types, but by default you'll use an optimized set of types in the `oracle.jbo.domain` and `oracle.ord.im` packages. These types, shown in [Table 3–1](#), allow data accessed from the Oracle database to remain in its native, internal format. You will achieve better performance using the optimized data types provided by ADF Business Components by avoiding costly type conversions when they are not necessary. To work with string-based data, by default ADF Business Components uses the regular `java.lang.String` type.

Table 3–1 Basic Data Types in the `oracle.jbo.domain` and `oracle.ord.im` Packages

Data Type	Package	Represents
Number	<code>oracle.jbo.domain</code>	Any numerical data
Date	<code>oracle.jbo.domain</code>	Date with optional time
DBSequence	<code>oracle.jbo.domain</code>	Sequential integer assigned by a database trigger
RowID	<code>oracle.jbo.domain</code>	Oracle database ROWID
Timestamp	<code>oracle.jbo.domain</code>	Timestamp value
TimestampTZ	<code>oracle.jbo.domain</code>	Timestamp value with Timezone information
BFileDomain	<code>oracle.jbo.domain</code>	Binary File (BFILE) object
BlobDomain	<code>oracle.jbo.domain</code>	Binary Large Object (BLOB)
ClobDomain	<code>oracle.jbo.domain</code>	Character Large Object (CLOB)

Table 3–1 (Cont.) Basic Data Types in the oracle.jbo.domain and oracle.ord.im Packages

Data Type	Package	Represents
OrdImageDomain	oracle.ord.im	Oracle Intermedia Image (ORDIMAGE)
OrdAudioDomain	oracle.ord.im	Oracle Intermedia Audio (ORDAUDIO)
OrdVideoDomain	oracle.ord.im	Oracle Intermedia Video (ORDVIDEO)
OrdDocDomain	oracle.ord.im	Oracle Intermedia Document (ORDDOC)
Struct	oracle.jbo.domain	User-defined object type
Array	oracle.jbo.domain	User-defined collection type (e.g. VARRAY)

Note: The `oracle.jbo.domain.Number` class has the same class name as the built-in `java.lang.Number` type. Since the Java compiler *implicitly* imports `java.lang.*` into every class, you need to explicitly import the `oracle.jbo.domain.Number` class into any class that references it. Typically, JDeveloper will follow this practice for you, but when you begin to write more custom code of your own, you'll learn to recognize compiler or runtime errors related to "Number is an abstract class" as indicating that you are inadvertently using `java.lang.Number` instead of `oracle.jbo.domain.Number`. Adding the:

```
import oracle.jbo.domain.Number;
```

line at the top of your class, after the package line, prevents these kinds of errors.

3.5.8 Generic Versus Strongly-Typed APIs

When working with application modules, view objects, and entity objects, you can choose to use a set of generic APIs or you can have JDeveloper generate code into a custom Java class to enable a strongly-typed API for that component. For example, when working with a view object, if you wanted to access the value of an attribute in any row of its result, the generic API would look like this:

```
Row row = serviceRequestVO.getCurrentRow();
Date requestDate = (Date)row.getAttribute("RequestDate");
```

Notice that using the generic APIs, you pass string names for parameters to the accessor, and you have to cast the return type to the expected type, as with `Date` shown in the example.

Alternatively, when you enable the strongly typed style of working you can write code like this:

```
ServiceRequestsRow row = (ServiceRequestRow)serviceRequestVO.getCurrentRow();
Date requestDate = row.getRequestDate();
```

In this case, you work with generated method names whose return type is known at compile time, instead of passing string names and having to cast the results. Typically, it is necessary to use strongly typed accessors when you need to invoke the methods from the business logic code without sacrificing compile-time safety. This can also be useful when you are writing custom validation logic in setter methods, although in this case, you may want to consider using Groovy expressions instead of generating entity and view row implementation classes for Business Components. Subsequent

chapters explain how to enable this strongly typed style of working by generating Java classes for business logic that you choose to implement using Java.

3.5.9 Custom Interface Support for Client-Accessible Components

Only these components of the business service are visible to the client:

- Application module, representing the service itself
- View objects, representing the query components
- View rows, representing each row in a given query component's results

The entity objects in the business service implementation are intentionally not designed to be referenced directly by clients. Instead, clients work with the data queried by view objects as part of an application module's data model. Behind the scenes, the view object cooperates automatically with entity objects in the business domain layer to coordinate validating and saving data that the user changes. For more information about this runtime interaction, see [Section 6.3.8, "What Happens at Runtime: When View Objects and Entity Objects Cooperate"](#).

3.5.9.1 Framework Client Interfaces for Components

The Java interfaces of the `oracle.jbo` package provide a client-accessible API for your business service. This package intentionally does not contain an `Entity` interface, or any methods that would allow clients to directly work with entity objects. Instead, client code works with interfaces like:

- `ApplicationModule`, to work with the application module
- `ViewObject`, to work with the view objects
- `Row`, to work with the view rows

3.5.9.2 Custom Client Interfaces for Components

When you begin adding custom code to your Oracle ADF business components that you want clients to be able to call, you can "publish" that functionality to clients for any client-visible component. For each of your components that publishes at least one custom method to clients on its client interface, JDeveloper automatically maintains the related Java interface file. So, assuming you were working with an application module like `StoreServiceAM` in the Fusion Order Demo application, you could have custom interfaces like:

- Custom application module interface
`StoreServiceAM extends ApplicationModule`
- Custom view object interface
`OrderItemsInfo extends ViewObject`
- Custom view row interface
`OrderItemsInfoRowClient extends Row`

Client code can then cast one of the generic client interfaces to the more specific one that includes the selected set of client-accessible methods you've selected for your particular component.

3.6 Overview of Groovy Support

Groovy is a scripting language for the Java platform with Java-like syntax. Because it is dynamically compiled, Groovy script can be stored inline in XML files to enable customization after deployment. And the Groovy language simplifies the authoring of code by employing dot-separated notation.

For example, with Groovy you can use simplified syntax like:

```
PromotionDate > HireDate
```

rather than:

```
((Date)getAttribute("PromotionDate")).compareTo((Date)getAttribute("HireDate")) > 0
```

in a validation rule.

Note that the current object is passed in to the script as the `this` object, so you can reference an attribute in the current object by simply using the attribute name. For example, in an attribute-level or entity-level Script Expression validator, to refer to an attribute named "HireDate", the script can simply reference `HireDate`.

There is one top-level object named `adf` that allows you access to objects that the framework makes available to the Groovy script. The accessible Oracle ADF objects consist of the following:

- `adf.context` - to reference the `ADFContext` object
- `adf.object` - to reference the object on which the expression is being applied (which can also be referenced using the keyword `object`, without the `adf` prefix).
- `adf.error` - in validation rules, to access the error handler that allows the validation expression to generate exceptions or warnings

You can also reference the current date (time truncated) or current date and time using the following expressions:

- `adf.currentDate`
- `adf.currentTime`

Other accessible member names come from the context in which the Groovy 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 variable participates.
- Transient attribute: The context is the current entity or view row. You can reference attributes by name in the entity or view row in which the attribute appears, as well as public methods on that entity or view row. To access methods on the current object, you must use the `object` keyword to reference the current object (for example, `object.methodName()`). The `object` keyword is equivalent to the `this` keyword in Java. Without it, in transient expressions, the method will be assumed to exist on the dynamically compiled Groovy script object itself.
- Script Expression validation rule: The context is the validator object (`JboValidatorContext`) merged with the entity on which the validator is applied. You can reference keywords like the following:
 - `newValue`: in an attribute-level validator, to access the attribute value being set

- `oldValue`: in an attribute-level validator, to access the current value of the attribute being set
- `source`: to access the entity on which the validator is applied in order to invoke methods on it (accessing the value of an attribute in the entity does not require the `source` keyword)

You can use the following built-in aggregate functions on ADF RowSet objects:

- `rowSetAttr.sum(GroovyExpr)`
- `rowSetAttr.count(GroovyExpr)`
- `rowSetAttr.avg(GroovyExpr)`
- `rowSetAttr.min(GroovyExpr)`
- `rowSetAttr.max(GroovyExpr)`

These aggregate functions accept a string-value argument that is interpreted as a Groovy expression that is evaluated in the context of each row in the rowset as the aggregate is being computed. The Groovy expression must return a numeric value (or number domain). For example:

```
employeesInDept.sum("Sal")
```

or

```
employeesInDept.sum("Sal!=0?Sal:0 + Comm!=0?Comm:0")
```

The ADF Business Components framework provides support for the use of Groovy to perform the following tasks:

- Define an Script Expression validator or Compare validator (see [Section 7.5, "Using Groovy Expressions For Validation and Business Rules"](#))
- Define error message tokens for handling validation failure (see [Section 7.7.4, "How to Embed a Groovy Expression in an Error Message"](#))
- Handle conditional execution of validators (see [Section 7.7.3, "How to Conditionally Raise Error Messages Using Groovy"](#))
- Set the default value of a bind variable in the view object query statement (see [Section 5.9, "Working with Bind Variables"](#))
- Set the default value of a bind variable that specifies a criteria item in the view criteria statement (see [Section 5.10, "Working with Named View Criteria"](#)).
- Define the default value for an entity object attribute (see [Section 4.10.6, "How to Define a Static Default Value"](#))
- Calculate the value of a transient attribute of an entity object or view object (see [Section 4.13, "Adding Transient and Calculated Attributes to an Entity Object"](#) and [Section 5.13, "Adding Calculated and Transient Attributes to a View Object"](#))

For more information about the Groovy language, refer to the following websites:

- <http://groovy.codehaus.org/>
- <http://java.sun.com/developer/technicalArticles/JavaLP/groovy/>

4

Creating a Business Domain Layer Using Entity Objects

This chapter describes how to use entity objects to create a reusable layer of business domain objects for use in your Java EE applications.

This chapter includes the following sections:

- [Section 4.1, "Introduction to Entity Objects"](#)
- [Section 4.2, "Creating Entity Objects and Associations"](#)
- [Section 4.3, "Creating and Configuring Associations"](#)
- [Section 4.4, "Creating an Entity Diagram for Your Business Layer"](#)
- [Section 4.5, "Defining Property Sets"](#)
- [Section 4.6, "Defining Attribute Control Hints for Entity Objects"](#)
- [Section 4.7, "Working with Resource Bundles"](#)
- [Section 4.8, "Defining Business Logic Groups"](#)
- [Section 4.9, "Configuring Runtime Behavior Declaratively"](#)
- [Section 4.10, "Setting Attribute Properties"](#)
- [Section 4.11, "Working Programmatically with Entity Objects and Associations"](#)
- [Section 4.12, "Generating Custom Java Classes for an Entity Object"](#)
- [Section 4.13, "Adding Transient and Calculated Attributes to an Entity Object"](#)

4.1 Introduction to Entity Objects

An entity object is the ADF Business Components component that represents a row in the specified data source and simplifies modifying its associated attributes.

Importantly, it allows you to encapsulate domain business logic to ensure that your business policies and rules are consistently validated.

Entity objects support numerous declarative business logic features to enforce the validity of your data. You will typically complement declarative validation with additional custom application logic and business rules to cleanly encapsulate a maximum amount of domain business logic into each entity object. Your associated set of entity objects forms a reusable business domain layer that you can exploit in multiple applications.

The key concepts of entity objects are the following:

- You define an entity object by specifying the database table whose rows it will represent.
- You can create associations to reflect relationships between entity objects.
- At runtime, entity rows are managed by a related entity definition object.
- Each entity row is identified by a related row key.
- You retrieve and modify entity rows in the context of an application module that provides the database transaction.

4.2 Creating Entity Objects and Associations

If you already have a database schema to work from, the simplest way to create entity objects and associations is to reverse-engineer them from existing tables. When needed, you can also create an entity object from scratch, and then generate a table for it later.

4.2.1 How to Create Multiple Entity Objects and Associations from Existing Tables

To create one or more entity objects, use the Business Components from Tables wizard, which is available from the New Gallery.

To create one or more entity objects and associations from existing tables:

1. In the Application Navigator, right-click the project in which you want to create the entity objects and choose **New**.
2. In the New Gallery, expand **Business Tier**, select **ADF Business Components**, and then select **Business Components from Tables** and click **OK**.
If this is the first component you're creating in the project, the Initialize Business Components Project dialog appears to allow you to select a database connection.
3. In the Initialize Business Components Project dialog, select the database connection or choose **New** to create a connection. Click **OK**.
4. On the Entity Objects page, do the following to create the entity objects:
 - Enter the package name in which all of the entity objects will be created.
 - Select the tables from the **Available** list for which you want to create entity objects.

If the **Auto-Query** checkbox is selected, then the list of available tables appears immediately. In the **Name Filter** field, you can optionally enter a full or partial table name to filter the available tables list in real time. As an alternative to the auto-query feature, click the **Query** button to retrieve the list based on an optional table name filter. When no name filter is entered, JDeveloper retrieves all table objects for the chosen schema.

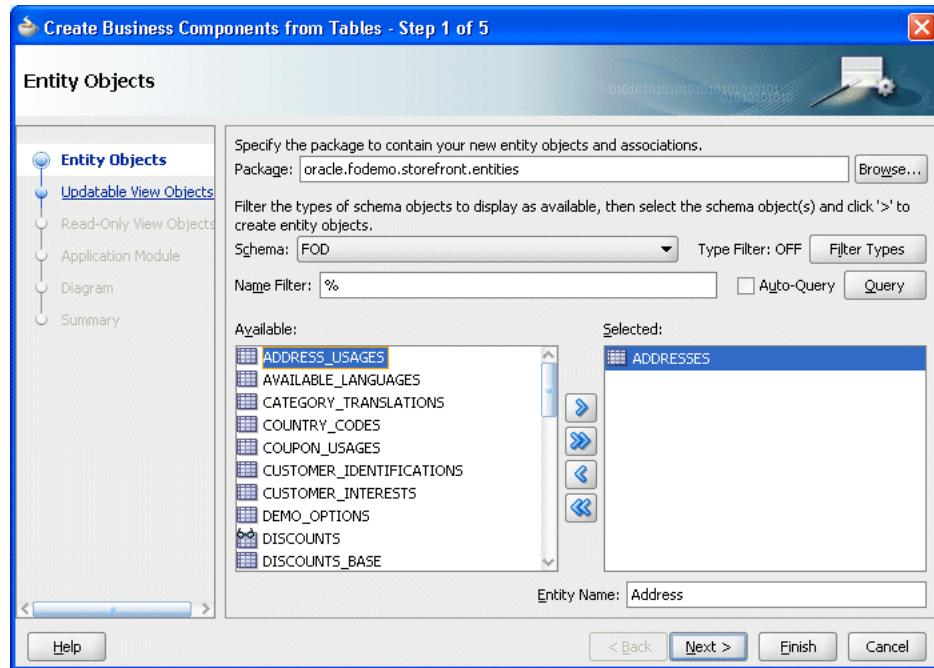
- Click **Filter Types** if you want to see only a subset of the database objects available. You can filter out tables, views, or synonyms.

Once you have selected a table from the **Available** list, the proposed entity object name for that table appears in the **Selected** list with the related table name in parenthesis.

- Select an entity object name in the **Selected** list and use the **Entity Name** field to change the default entity object name.

Best Practice: Since each entity object instance represents a *single* row in a particular table, name the entity objects with a singular noun (like Address, Order, and Person), instead of their plural counterparts. [Figure 4–1](#) shows what the wizard page looks like after selecting the ADDRESSES table in the FOD schema, setting a package name of oracle.fodemo.storefront.entities, and renaming the entity object in the singular.

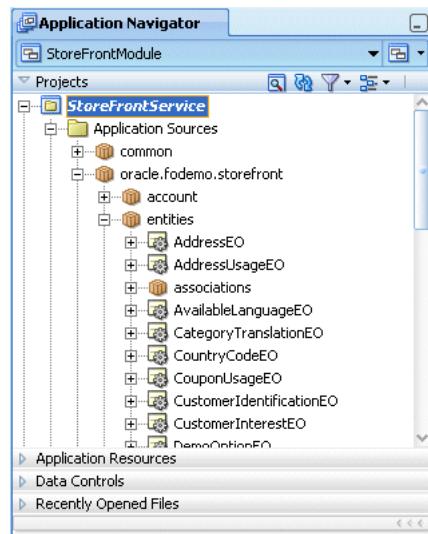
[Figure 4–1 Create Business Components from Tables Wizard, Entity Objects Page](#)



- When you are satisfied with the selected table objects and their corresponding entity object names, click **Finish**.

The Application Navigator displays the entity objects in the package you specified.

Best Practice: After you create associations, move all of your associations to a separate package so you can view and manage them separately from the entity objects. In [Figure 4–2](#), the associations have been moved to a subpackage (associations) and do not appear in the entities package in the Application Navigator. For more information, see [Section 4.3.4, "How to Rename and Move Associations to a Different Package"](#).

Figure 4–2 New Entity Objects in Application Navigator

4.2.2 How to Create Single Entity Objects Using the Create Entity Wizard

To create a single entity object, you can use the Create Entity Object wizard, which is available in the New Gallery.

To create a single entity object and association:

1. In the Application Navigator, right-click the project in which you want to create the entity object and choose **New**.
2. In the New Gallery, expand **Business Tier**, select **ADF Business Components**, and then select **Entity Object** and click **OK**.
If this is the first component you're creating in the project, the Initialize Business Components Project dialog appears to allow you to select a database connection.
3. In the Initialize Business Components Project dialog, select the database connection or choose **New** to create a connection. Click **OK**.
4. On the Name page, do the following to create the entity object:
 - Enter the package name in which the entity object will be created.
 - Click **Browse** (next to the **Schema Object** field) to select the table for which you want to create the entity object.
Or, if you plan to create the table later, you can enter a name of a table that does not exist.
5. If you manually entered a table name in the **Schema Objects** field, you will need to define each attribute on the Attributes page of the wizard. Click **Next**.
You can create the table manually or generate it, as described in [Section 4.2.6, "How to Create Database Tables from Entity Objects"](#).
6. When you are satisfied with the table object and its corresponding entity object name, click **Finish**.

4.2.3 What Happens When You Create Entity Objects and Associations from Existing Tables

When you create an entity object from an existing table, first JDeveloper interrogates the data dictionary to infer the following information:

- The Java-friendly entity attribute names from the names of the table's columns (for example, USER_ID -> UserId)
- The SQL and Java data types of each attribute based on those of the underlying column
- The length and precision of each attribute
- The primary and unique key attributes
- The mandatory flag on attributes, based on NOT NULL constraints
- The relationships between the new entity object and other entities based on foreign key constraints

Note: Since an entity object represents a database row, it seems natural to call it an entity row. Alternatively, since at runtime the entity row is an instance of a Java object that encapsulates business logic for that database row, the more object-oriented term *entity instance* is also appropriate. Therefore, these two terms are interchangeable.

JDeveloper then creates the XML component definition file that represents its declarative settings and saves it in the directory that corresponds to the name of its package. For example, when an entity named Order appears in the genericbcmodel.entities package, JDeveloper will create the XML file genericbcmodel/entities/Order.xml under the project's source path. This XML file contains the name of the table, the names and data types of each entity attribute, and the column name for each attribute.

You can inspect the XML description for the entity object by selecting the object in the Application Navigator and looking in the corresponding **Source** pane in the overview editor.

Note: If your IDE-level Business Components Java generation preferences so indicate, the wizard may also create an optional custom entity object class (for example, OrderImpl.java).

4.2.3.1 What Happens When Tables Have Foreign Key Relationships

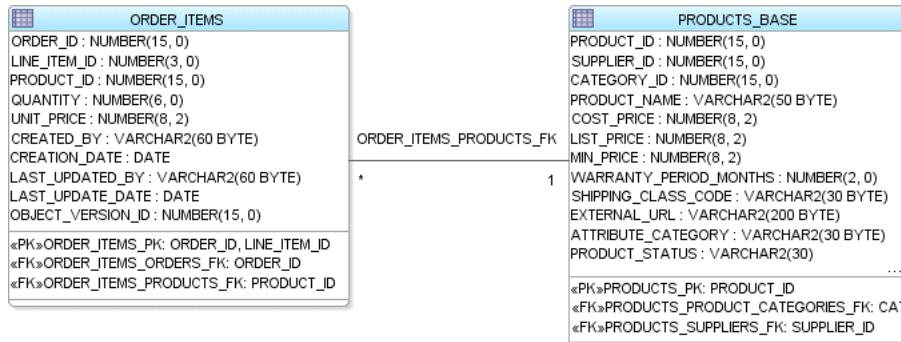
In addition to the entity objects, the wizard also generates named association components that capture information about the relationships between entity objects. For example, the database diagram in [Figure 4-3](#) shows that JDeveloper derives default association names like OrderItemsProductsFkAssoc by converting the foreign key constraint names to a Java-friendly name and adding the Assoc suffix. For each association created, JDeveloper creates an appropriate XML component definition file and saves it in the directory that corresponds to the name of its package.

By default the associations reverse-engineered from foreign keys are created in the same package as the entities. For example, for the association OrderItemsProductsFkAssoc with entities in the fodemo.storefront.entities package, JDeveloper creates the association XML

file named

`./fodemo/storefront/entities/OrderItemsProductsFkAssoc.xml`.

Figure 4–3 ORDER_ITEMS and PRODUCTS_BASE Tables Related by Foreign Key



4.2.3.2 What Happens When a Table Has No Primary Key

If a table has no primary key constraint, then JDeveloper cannot infer the primary key for the entity object. Since every entity object must have at least one attribute marked as a primary key, the wizard will create an attribute named `RowID` and use the database `ROWID` value as the primary key for the entity. If appropriate, you can edit the entity object later to mark a different attribute as a primary key and remove the `RowID` attribute. When you use the Create Entity Object wizard and you have not set any other attribute as primary key, you will be prompted to use `RowID` as the primary key.

4.2.4 What Happens When You Create an Entity Object for a Synonym or View

When you create an entity object using the Business Components from Tables wizard or the Create Entity Object wizard, the object can represent an underlying table, synonym, or view. The framework can infer the primary key and related associations for a table or synonym by inspecting database primary and foreign key constraints in the data dictionary.

However, when your selected schema object is a database view, then neither the primary key nor associations can be inferred since database views do not have database constraints. In this case, if you use the Business Components from Tables wizard, the primary key defaults to `RowID`. If you use the Create Entity Object wizard, you'll need to specify the primary key manually by marking at least one of its attributes as a primary key. For more information, see [Section 4.2.3.2, "What Happens When a Table Has No Primary Key"](#).

When your selected schema object is a synonym, then there are two possible outcomes. If the synonym is a synonym for a table, then the wizard and editor will behave as if you had specified a table. If instead the synonym refers to a database view, then they will behave as if you had specified a view.

4.2.5 How to Edit an Existing Entity Object or Association

After you've created a new entity object or association, you can edit any of its settings in the overview editor. To launch the editor, choose **Open** from the context menu for the entity object or association in the Application Navigator or double-click on the object. By clicking on the different tabs of the editor, you can adjust the settings that define the object and govern its runtime behavior.

4.2.6 How to Create Database Tables from Entity Objects

To create database tables based on entity objects, right-click the package in the Application Navigator that contains the entity objects and choose **Create Database Objects** from the context menu. A dialog appears to let you select the entities whose tables you'd like to create. This tool can be used to generate a table for an entity object you created from scratch, or to drop and re-create an existing table.

Caution: This feature does *not* generate a DDL script to run later, it performs its operations directly against the database and will drop existing tables. A dialog appears to confirm that you want to do this before proceeding. For entities based on existing tables, use with caution.

In the overview editor for an association, the **Use Database Key Constraints** checkbox on the **Association Properties** page controls whether the related foreign key constraint will be generated when creating the tables for entity objects. Selecting this option does not have any runtime implications.

4.2.7 How to Synchronize an Entity with Changes to Its Database Table

Inevitably you (or your DBA) might alter a table for which you've already created an entity object. Your existing entity will not be disturbed by the presence of additional attributes in its underlying table; however, if you want to access the new column in the table in your Java EE application, you'll need to synchronize the entity object with the database table.

For example, suppose you had done the following at the SQL*Plus command prompt to add a new SECURITY_QUESTION column to the PERSONS table:

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

Then you can use the synchronization feature to add the new column as an attribute on the entity object.

To synchronize an entity with changes to its database table:

1. In the Application Navigator, right-click the desired entity object and choose **Synchronize with Database** from the context menu.

The Synchronize with Database dialog shows the list of the actions that can be taken to synchronize the business logic tier with the database.

2. Select the action you want to take.
 - Select one or more actions from the list, and click **Synchronize** to synchronize the selected items.
 - Click **Synchronize All** to perform all actions in the list.
 - Click **Write to File** to save the action list to a text file. This feature helps you keep track of the changes you make.
3. When finished, click **OK** to close the dialog.

4.2.7.1 Removing an Attribute Associated with a Dropped Column

The synchronize feature does not handle dropped columns. When a column is dropped from the underlying database after an entity object has been created, you can

delete the corresponding attribute from the entity object. If the attribute is used in other parts of your application, you must remove those usages as well.

To remove an entity attribute:

1. In the Application Navigator, double-click the entity to open it in the overview editor.
2. Click the Attributes tab to display the entity's attributes.
3. Right-click on the attribute, and choose **Delete Safely** from the context menu.
If there are other usages, the Delete Attributes dialog displays the message "Usages were found."
4. Click **View Usages**.
The Log window shows all usages of the attribute.
5. Work through the list in the Log window to delete all usages of the entity attribute.

4.2.7.2 Addressing a Data Type change in the Underlying Table

The synchronize feature does not handle changed data types. For a data type change in the underlying table (for example, precision increased), you must locate all usages of the attribute and manually make changes, as necessary.

To locate all usages of an entity attribute:

1. In the Application Navigator, double-click the entity to open it in the overview editor.
2. Click the Attributes tab to display the entity's attributes.
3. Right-click on the attribute, and choose **Find Usages** from the context menu.
If there are other usages, they are displayed in the Log window.

4.2.8 How to Store Data Pertaining to a Specific Point in Time

Effective dated tables are used to provide a view into the data set pertaining to a specific point in time. Effective dated tables are widely used in applications like HRMS and Payroll to answer queries like:

- What was the tax rate for an employee on August 31st, 2005?
- What are the employee's benefits as of October 2004?

In either case, the employee's data may have changed to a different value since then.

The primary difference between the effective dated entity type and the dated entity type is that the dated entity does not cause row splits during update and delete.

When you create an effective dated entity object, you identify the entity as effective dated and specify the attributes of the entity that represent the start and end dates. The start date and end date attributes must be of the Date type.

Additionally, you can specify an attribute that represents the sequence for the effective dated entity and an attribute that represents a flag for the sequence. These attributes allow for tracking of multiple changes in a single day.

To create an effective dated entity object

1. In the Application Navigator, double-click the entity on which you want enable effective dating to open it in the overview editor.

2. In the Property Inspector, expand the **Type** category.

If necessary, choose **Property Inspector** from the **View** menu to display the Property Inspector.

If the **Type** category is not displayed in the Property Inspector, click the **General** tab in the overview editor to set the proper focus.

3. From the context menu for the **Effective Date Type** property, choose **Edit**.

To display the context menu, click the down arrow next to the property field.

4. In the Edit Property dialog, specify the following settings:

- For **Effective Date Type**, select **EffectiveDated**.
- For **Start Date Attribute**, select the attribute that corresponds to the start date.
- For **End Date Attribute**, select the attribute that corresponds to the end date.
- 5. You can optionally specify attributes that allow for tracking of multiple changes in a single day.
 - For **Effective Date Sequence**, select the attribute that stores the sequence of changes.
 - For **Effective Date Sequence Flag**, select the attribute that stores a flag indicating the most recent change in the sequence.

Without specifying the **Effective Date Sequence** and **Effective Date Sequence Flag** attributes, the default granularity of effective dating is one day. For this reason, multiple changes in a single day are not allowed. An attempt to update the entity a second time in a single day will result in an exception being thrown. After these two attributes are specified, the framework inserts and updates their values as necessary to track multiple changes in a single day.

6. Click **OK**.

Note: You can also identify the start and end date attributes using the Property Inspector for the appropriate attributes. To do so, select the appropriate attribute in the overview editor and set the **IsEffectiveStartDate** or **IsEffectiveEndDate** property to true in the Property Inspector.

4.2.9 What Happens When You Create Effective Dated Entity Objects

When you create an effective dated entity object, JDeveloper creates a transient attribute called **SysEffectiveDate** to store the effective date for the row. Typically the Insert, Update, and Delete operations modify the transient attribute while the ADF framework decides the appropriate values for Effective Start Date and Effective End Date.

[Example 4–1](#) show some sample XML entries that are generated when you create an effective dated entity. For more information about working with effective dated objects, see [Section 5.4, "Limiting View Object Rows Using Effective Date Ranges"](#).

Example 4–1 XML Entries for Effective Dated Entities

```
// In the effective dated entity
<Entity>
  ...

```

```
EffectiveDateType="EffectiveDated">

// In the attribute identified as the start date
<Attribute
...
    IsEffectiveStartDate="true">

// In the attribute identified as the end date
<Attribute
...
    IsEffectiveEndDate="true">

// The SysEffectiveDate transient attribute
<Attribute
    Name="SysEffectiveDate"
    IsQueriable="false"
    IsPersistent="false"
    ColumnName="$none$"
    Type="oracle.jbo.domain.Date"
    ColumnType="$none$"
    SQLType="DATE"/>
```

4.2.10 What You May Need to Know About Creating Entities from Tables

The Business Components from Tables wizard makes it easy to quickly generate many business components at the same time. In practice, this does not mean that you should use it to immediately create entity objects for every table in your database schema just because it is possible to do so. If your application requires all of the tables, then that strategy might be appropriate. But because you can use the wizard whenever needed, you should create the entity objects for the tables that you know will be involved in the application.

[Section 9.4, "Defining Nested Application Modules"](#) describes a use case-driven design approach for your business services that can assist you in understanding which entity objects are required to support your application's business logic needs. You can always add more entity objects later as necessary.

4.3 Creating and Configuring Associations

If your database tables have no foreign key constraints defined, JDeveloper won't be able to infer the associations between the entity objects that you create. Since several ADF Business Components runtime features depend on the presence of entity associations, create them manually if the foreign key constraints don't exist.

4.3.1 How to Create an Association

To create an association, use the Create New Association wizard, which is available in the New Gallery.

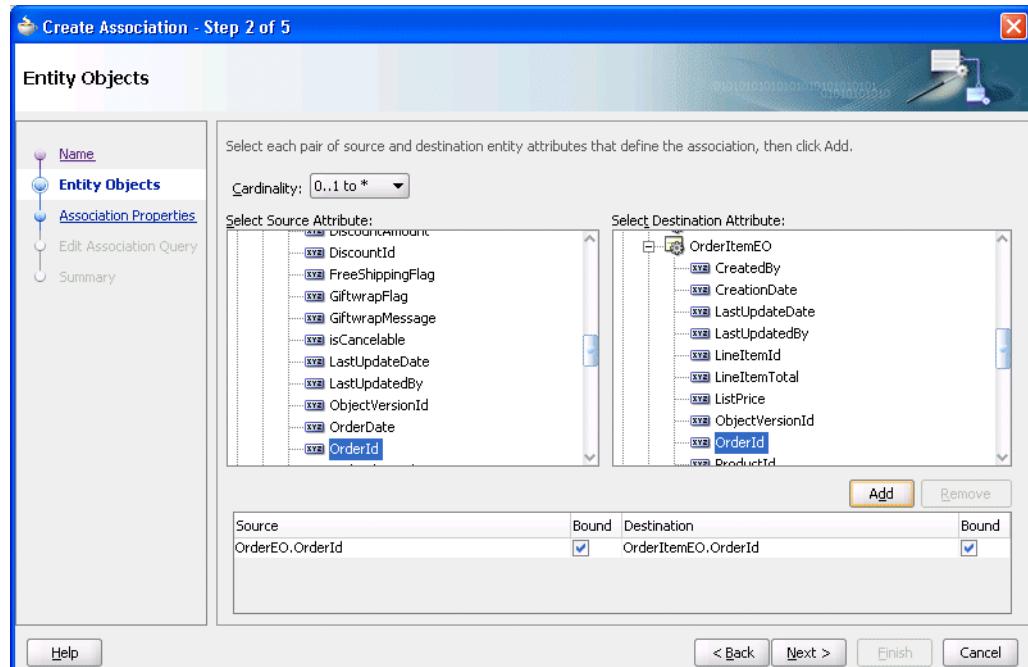
To create an association:

1. In the Application Navigator, right-click the project in which you want to create the association and choose **New**.
2. In the New Gallery, expand **Business Tier**, select the **ADF Business Components** and then select **Association** and click **OK**.
3. On the Name page, do the following to create the association:

- Enter the package name in which the association will be created.
 - Enter the name of the association component.
 - Click **Next**.
4. On the Entity Objects page, select the source and destination entity attributes:
- Select a source attribute from one of the entity objects that is involved in the association to act as the master.
 - Select a corresponding destination attribute from the other entity object involved in the association.

For example, [Figure 4–4](#) shows the selected OrderId attribute from the OrderEO entity object as the source entity attribute. Because the OrderItemEO rows contain an order ID that relates them to a specific OrderEO row, you would select this OrderId foreign key attribute in the OrderItemEO entity object as the destination attribute.

Figure 4–4 Create Association Wizard, Attribute Pairs That Relate Two Entity Objects Defined



5. Click **Add** to add the matching attribute pair to the table of source and destination attribute pairs below.

By default, the **Bound** checkbox is selected for both the source and destination attribute. This checkbox allows you to specify whether or not the value will be bound into the association SQL statement that is created internally when navigating from source entity to target entity or from target entity to source entity (depending on which side you select).

Typically, you would deselect the checkbox for an attribute in the relationship that is a transient entity attribute whose value is a constant and therefore should not participate in the association SQL statement to retrieve the entity.

6. If the association requires multiple attribute pairs to define it, you can repeat the preceding steps to add additional source/target attribute pairs.

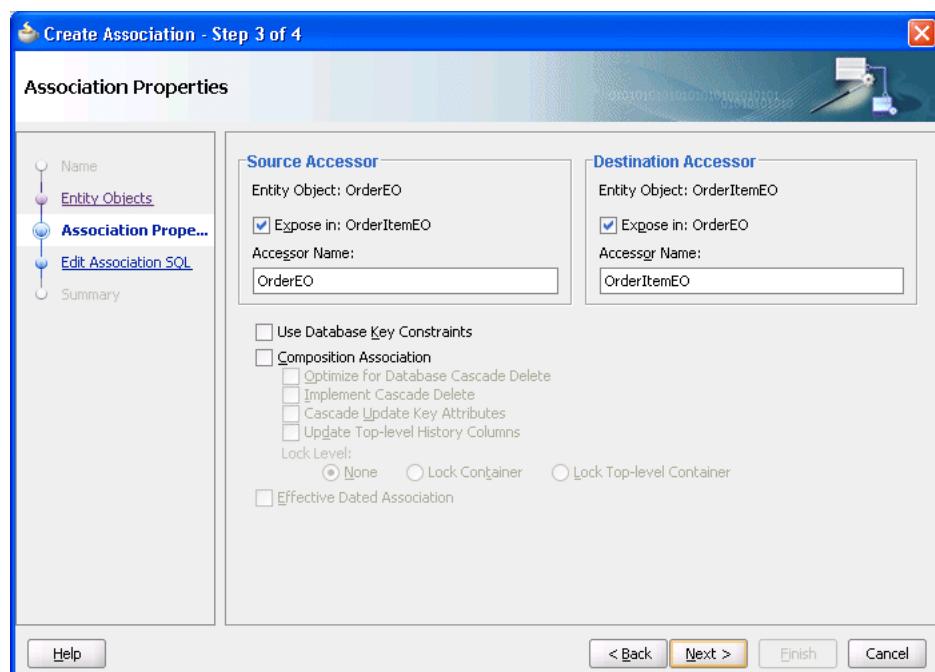
- Finally, ensure that the **Cardinality** dropdown correctly reflects the cardinality of the association. The default is a one-to-many relationship. Click **Next**.

For example, since the relationship between a `OrderEO` row and its related `OrderItemEO` rows is one-to-many, you can leave the default setting.

- On the Association SQL page, you can preview the association SQL predicate that will be used at runtime to access the related destination entity objects for a given instance of the source entity object.
- On the Association Properties page, disable the **Expose Accessor** checkbox on either the **Source** or the **Destination** entity object when you want to create an association that represents a one-way relationship. The default, bidirectional navigation is more convenient for writing business validation logic, so in practice, you typically leave these default checkbox settings.

For example, [Figure 4-5](#) shows an association that represents a bidirectional relationship, permitting either entity object to access the related entity row(s) on the other side when needed. In this example, this means that if you are working with an instance of an `OrderEO` entity object, you can easily access the collection of its related `OrderItemEO` rows. With any instance of a `OrderItemEO` entity object, you can also easily access the `Order` to which it belongs.

Figure 4-5 Association Properties Control Runtime Behavior



- When you are satisfied with the association definition, click **Finish**.

4.3.2 What Happens When You Create an Association

When you create an association, JDeveloper creates an appropriate XML component definition file and saves it in the directory that corresponds to the name of its package. For example, if you created an association named `OrderItemsOrdersFkAssoc` in the `oracle.fodemo.storefront.entities.associations` subpackage, then the association XML file would be created in the
`./oracle/fodemo/storefront/entities/associations` directory with the

name `OrderItemsOrdersFkAssoc.xml`. At runtime, the entity object uses the association information to automate working with related sets of entities.

4.3.3 How to Change Entity Association Accessor Names

You should consider the default settings for the accessor names on the Association Properties page and decide whether changing the names to something more intuitive is appropriate. The default settings define the names of the accessor attributes you will use at runtime to programmatically access the entities on the other side of the relationship. By default, the accessor names will be the names of the entity object on the other side. Since the accessor names on an entity must be unique among entity object attributes and other accessors, if one entity is related to another entity in *multiple* ways, then the default accessor names are modified with a numeric suffix to make the name unique.

In an existing association, you can rename the accessor using the Association Properties dialog.

To rename the entity accessor in an association:

1. Open the association in the overview editor.
2. On the **Relationships** page, expand the **Accessors** category and click the **Edit** icon. The Association Properties dialog displays the current settings for the associations accessors.
3. Make changes as necessary, and click **OK** to apply your changes and close the dialog.

4.3.4 How to Rename and Move Associations to a Different Package

Since associations are a component that you typically configure at the outset of your project and don't change frequently thereafter, you might want to move the associations to a different package so that your entity objects are easier to see. Both renaming components and moving them to a different package is straightforward using JDeveloper's refactoring functionality.

To move a set of business components to a different package:

1. In the Application Navigator, select the components you want to move.
2. Right-click one of the selected components, and choose **Refactor > Move** from the context menu.
3. In the Move Business Components dialog, enter the name of the package to move the component(s) to, or click **Browse** to navigate to and select the package.
4. Click **OK** to apply your changes and close the dialog.

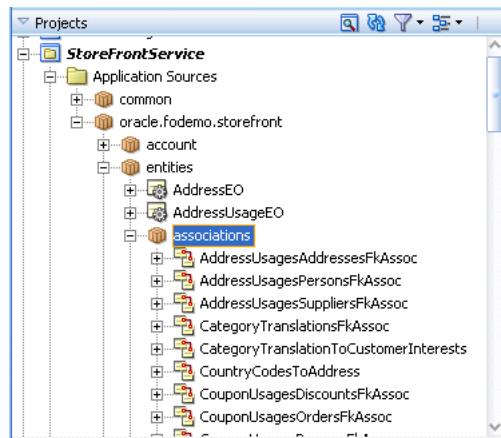
To rename a component:

1. In the Application Navigator, right-click the component you want to rename, and choose **Refactor > Rename** from the context menu.
2. In the Rename dialog, enter the new name for the component and click **OK**.

When you refactor ADF Business Components, JDeveloper moves the XML and Java files related to the components, and updates any other components that might reference them.

Figure 4–6 shows what the Application Navigator would look like after renaming all of the associations and moving them to the `oracle.fodemo.storefront.associations` subpackage. While you can refactor the associations into any package name you choose, picking a subpackage keeps them logically related to the entities, and allows you to collapse the package of associations to better manage which files display in the Application Navigator.

Figure 4–6 Application Navigator After Association Refactoring



4.3.5 What You May Need to Know About Using a Custom View Object in an Association

You can associate a custom view object with the source end or destination end (or both) of an entity association.

When you traverse entity associations in your code, if the entities are not already in the cache, then the ADF Business Components framework performs a query to bring the entity (or entities) into the cache. By default, the query performed to bring an entity into the cache is the find-by-primary-key query that selects values for all persistent entity attributes from the underlying table. If the application performs a lot of programmatic entity association traversal, you could find that retrieving all of the attributes might be heavy-handed for your use cases.

Entity associations support the ability to associate a custom, entity-based view object with the source entity or destination entity in the association, or both. The primary entity usage of the entity-based view object you supply must match the entity type of the association end for which you use it.

Using a custom view object can be useful because the custom view object's query can include fewer columns and it can include an `ORDER BY` clause. This allows you to control how much data is retrieved when an entity is brought into the cache due to association traversal, as well as the order in which any collections of related entities will appear.

For more information about creating a custom view object, see [Section 35.9.2, "How to Create an Entity-Based Programmatic View Object"](#).

4.3.6 What You May Need to Know About Composition Associations

A *composition association* represents a relationship between entities, such as `Person` referenced by an `Order` or a `OrderItem` contained in a `Order`. When you create

composition associations, it is useful to know about the kinds of relationships you can represent, and the various options.

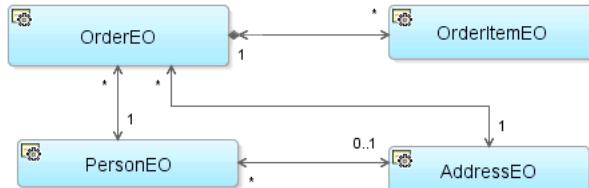
Associations between entity objects can represent two styles of relationships depending on whether the source entity:

- References the destination entity
- Contains the destination entity as a logical, nested part

[Figure 4-7](#) depicts an application business layer that represents both styles of relationships. For example, an OrderEO entry references a PersonEO. This relationship represents the first kind of association, reflecting that a PersonEO or an OrderEO entity object can exist independent from each other. In addition, the removal of an Order does not imply the cascade removal of the Person to which it was referring.

In contrast, the relationship between OrderEO and its collection of related OrderItemEO details is stronger than a simple reference. The OrderItemEO entries comprise a logical part of the overall OrderEO. In other words, a OrderEO is composed of OrderItemEO entries. It does not make sense for a OrderItemEO entity row to exist independently from an OrderEO, and when an OrderEO is removed — assuming the removal is allowed — all of its composed parts should be removed as well. This kind of logical containership represents the second kind of association, called a *composition*. The UML diagram in [Figure 4-7](#) illustrates the stronger composition relationship using the solid diamond shape on the side of the association which composes the other side of the association.

Figure 4-7 ServiceRequest Composed of ServiceHistory Entries and References Both Product and User



The Business Components from Tables Wizard creates composition associations by default for any foreign keys that have the ON DELETE CASCADE option. You can use the Create Association wizard or the overview editor for the association to indicate that an association is a composition association. Check the **Composition Association** checkbox on either the Association Properties page of the Create Association wizard or the Relationships page of the overview editor. An entity object offers additional runtime behavior in the presence of a composition. For the settings that control the behavior, see [Section 4.10.13, "How to Configure Composition Behavior"](#).

4.4 Creating an Entity Diagram for Your Business Layer

Since your layer of business domain objects represents a key reusable asset for your team, it is often convenient to visualize the business domain layer using a UML model. JDeveloper supports easily creating a diagram for your business domain layer that you and your colleagues can use for reference.

The UML diagram of business components is not just a static picture that reflects the point in time when you dropped the entity objects onto the diagram. Rather, it is a UML-based rendering of the current component definitions, that will always reflect the current state of affairs. What's more, the UML diagram is both a visualization aid

and a visual navigation and editing tool. To open the overview editor for any entity object in a diagram, right-click the desired object and choose **Properties** from the context menu or double-click the desired object. You can also perform some entity object editing tasks directly on the diagram, like renaming entities and entity attributes, and adding or removing attributes.

4.4.1 How to Create an Entity Diagram

To create a diagram of your entity objects, you can use the Create Business Components Diagram dialog, which is available in the New Gallery.

To create an entity diagram that models existing entity objects:

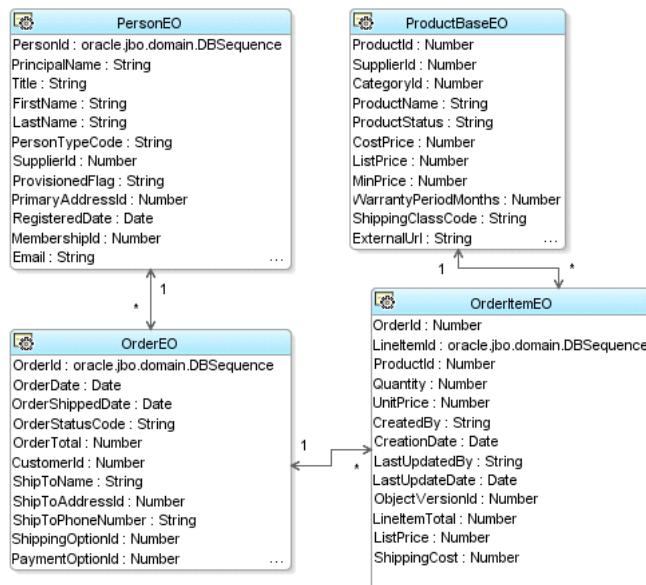
1. In the Application Navigator, right-click the project in which you want to create the entity diagram and choose **New**.
2. In the New Gallery, expand **Business Tier**, select **ADF Business Components**, then select **Business Components Diagram** and click **OK**.
3. In the dialog, do the following to create the diagram:
 - Enter a name for the diagram, for example **Business Domain Objects**.
 - Enter the package name in which the diagram will be created. For example, you might create it in a subpackage like **myproject.model.design**.
4. Click **OK**.
5. To add existing entity objects to the diagram, select them in the Application Navigator and drop them onto the diagram surface.

After you have created the diagram you can use the Property Inspector to adjust visual properties of the diagram. For example you can:

- hide or show the package name
- change the font
- toggle the grid and page breaks on or off
- display association names that may otherwise be ambiguous

You can also create an image of the diagram in **PNG**, **JPG**, **SVG**, or compressed **SVG** format, by choosing **Publish Diagram** from the context menu on the diagram surface.

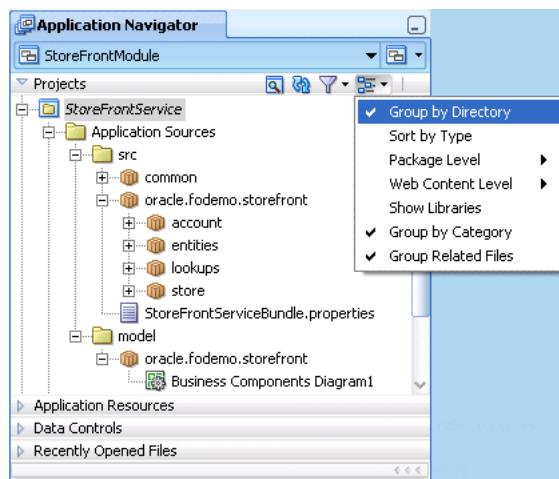
Figure 4–8 shows a sample diagram that models various entity objects from the business domain layer.

Figure 4–8 UML Diagram of Business Domain Layer

4.4.2 What Happens When You Create an Entity Diagram

When you create a business components diagram, JDeveloper creates an XML file `*.oxd_bc4j` representing the diagram in a subdirectory of the project's model path that matches the package name in which the diagram resides.

By default, the Application Navigator unifies the display of the project contents paths so that ADF components and Java files in the source path appear in the same package tree as the UML model artifacts in the project model path. However, as shown in [Figure 4–9](#), using the **Navigator Display Options** toolbar button on the Application Navigator, you can see the distinct project content path root directories when you prefer.

Figure 4–9 Toggling the Display of Separate Content Path Directories

4.4.3 What You May Need to Know About the XML Component Descriptors

When you include a business component like an entity object to a UML diagram, JDeveloper adds extra metadata to a <Data> section of the component's XML component descriptor as shown in [Example 4–2](#). This additional information is used at design time only.

Example 4–2 Additional UML Metadata Added to an Entity Object XML Descriptor

```
<Entity Name="OrderEO" ... >
  <Data>
    <Property Name ="COMPLETE_LIBRARY" Value ="FALSE" />
    <Property Name ="ID"
      Value ="ff16fca0-0109-1000-80f2-8d9081ce706f::::EntityObject" />
    <Property Name ="IS_ABSTRACT" Value ="FALSE" />
    <Property Name ="IS_ACTIVE" Value ="FALSE" />
    <Property Name ="IS_LEAF" Value ="FALSE" />
    <Property Name ="IS_ROOT" Value ="FALSE" />
    <Property Name ="VISIBILITY" Value ="PUBLIC" />
  </Data>
  :
</Entity>
```

4.4.4 What You May Need to Know About Changing the Names of Components

On an entity diagram, the names of entity objects, attributes, and associations can be changed for clarity. Changing names on a diagram does not affect the underlying data names. The name change persists for the diagram only. The new name may contain spaces and mixed case for readability. To change the actual entity object names, attribute names, or association names, open the entity object or association in the overview editor.

4.5 Defining Property Sets

A property set is a named collection of properties, where a property is defined as a name/value pair. Property sets are a convenience mechanism to group properties and then reference them from other ADF Business Components objects. Properties defined in a property set can be configured to be translatable, in which case the translations are stored in a message bundle file owned by the property set.

Property sets can be used for a variety of functions, such as control hints and error messages. A property set may contain control hints and other custom properties, and you can associate them with multiple attributes of different objects.

Note: Take care when defining property sets that contain translatable content. Be sure not to "overload" common terms in different contexts. For example, the term "Name" might be applied to both an object and a person in one language, but then translated into two different terms in a target language. Even though a term in several contexts might be the same in the source language, a separate distinguishable term should be used for each context.

Property sets can be used with entity objects and their attributes, view objects and their attributes, and application modules.

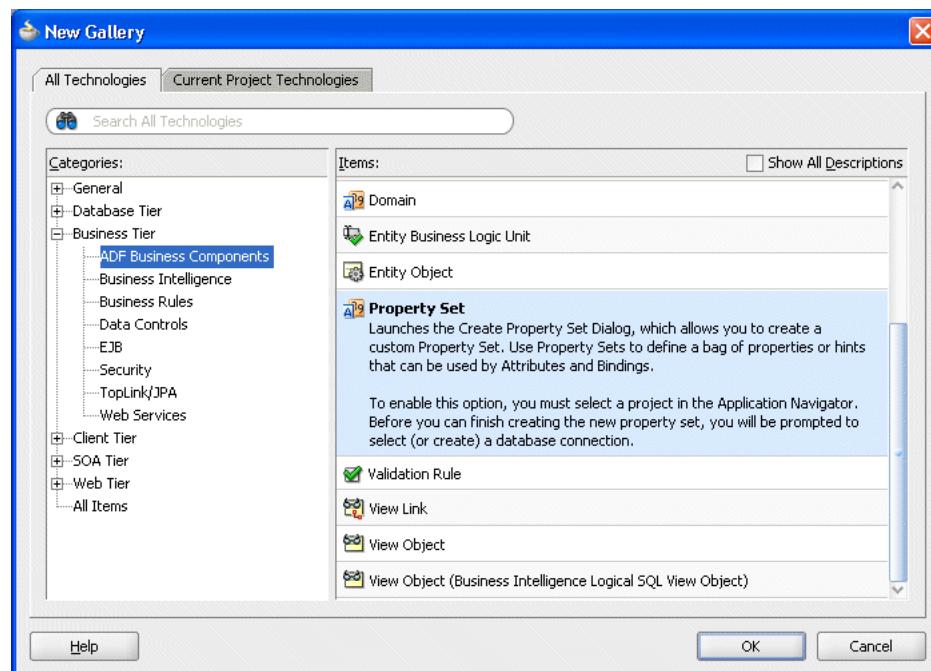
4.5.1 How to Define a Property Set

To define a property set, you create a new property set using a dialog and then specify properties using the Property Inspector.

To define a property set:

1. In the Application Navigator, right-click the project where you want to create the property set, and choose **New** from the context menu.
2. In the New Gallery, expand **Business Tier**, select **ADF Business Components**, then select **Property Set** and click **OK**.

Figure 4–10 Property Set in New Gallery



3. In the Create Property Set dialog, enter the name and location of the property set and click **OK**.
4. From the **View** menu, choose **Property Inspector**.
5. In the Property Inspector, define the properties for the property set.

4.5.2 How to Apply a Property Set

After you have created the property set, you can apply the property set to an entity object or attribute, and use the defined properties (or override them, if necessary).

To apply a property set to an entity object or view object:

1. Open the project that contains the entity object or view object you want to edit.
2. In the Application Navigator, double-click the desired object to open the overview editor.
3. In the overview editor, on the **General** tab, click the **Edit** icon next to the **Property Set** line.
4. Select the appropriate property set, and click **OK**.

To apply a property set to an attribute:

1. Open the project that contains the entity object or view object you want to edit.
2. In the Application Navigator, double-click the desired entity object to open the overview editor.
3. In the overview editor, click the **Attributes** tab, and double-click the attribute you want to edit.
4. In the Attribute editor, click the first node to view the general properties of the attribute.

For view objects, it is the **View Attribute** node. For entity objects, it is the **Entity Attribute** node.

5. In the **Property Set** dropdown list, select the appropriate property set, and click **OK**.

4.6 Defining Attribute Control Hints for Entity Objects

If you are familiar with previous versions of ADF business components, you may have used control hints. Control hints allow you to define label text, tooltip, and format mask hints for entity object attributes. The UI hints you define on your business domain layer are inherited by any entity-based view objects as well. You can also set additional control hints on view objects and application modules in a similar manner.

4.6.1 How to Add Attribute Control Hints

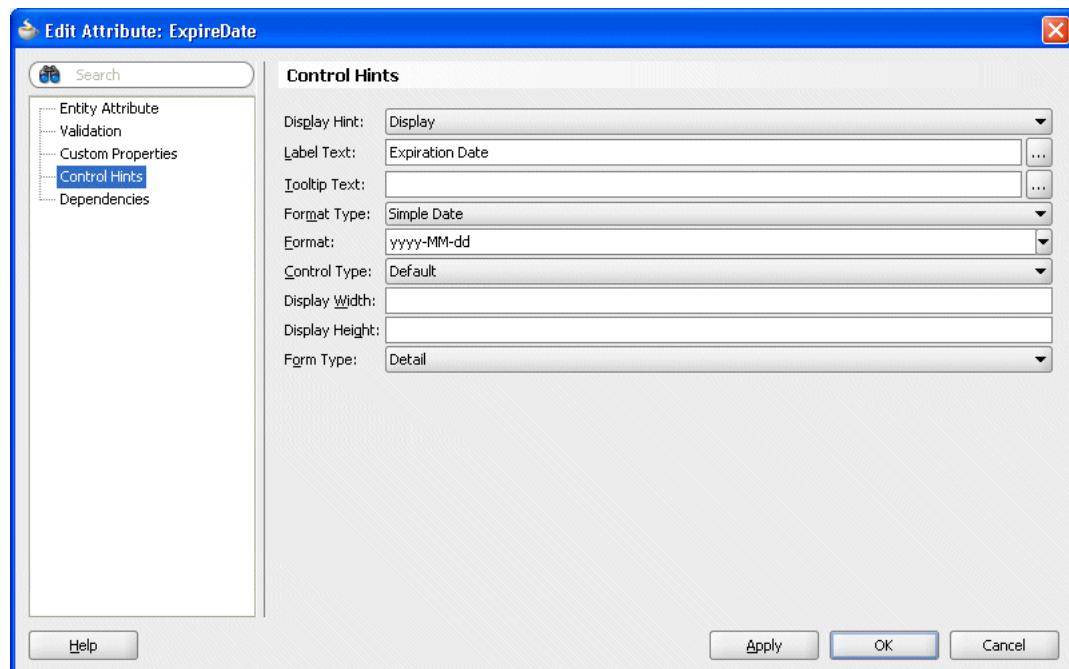
To add attribute control hints to an entity object, use the overview editor.

To add attribute control hints to an entity object:

1. Open the project that contains the entity object you want to edit.
2. In the Application Navigator, double-click the desired entity object to open the overview editor.
3. In the overview editor, click the **Attributes** tab, and double-click the attribute you want to edit.
4. In the Edit Attribute dialog, click the **Control Hints** node to view the attribute's control hints.
5. Specify control hints as necessary, and then click **OK**.

For example, [Figure 4-11](#) shows control hints defined for the attribute `ExpireDate` of the `PaymentOptionEO` entity object. The defined hints include the following:

- **Format Type** to Simple Date
- **Format mask** of `yyyy-MM-dd`

Figure 4–11 Edit Attribute Dialog, Control Hints Node

Note: Java defines a standard set of format masks for numbers and dates that are different from those used by the Oracle database's SQL and PL/SQL languages. For reference, see the Javadoc for the `java.text.DecimalFormat` and `java.text.SimpleDateFormat` classes.

4.6.2 What Happens When You Add Attribute Control Hints

When you define attribute control hints for an entity object, JDeveloper creates a resource bundle file in which to store them. The hints that you define can be used by generated forms and tables in associated view clients. The type of file and its granularity are determined by Resource Bundle options in the Project Properties dialog. For more information, see [Section 4.7, "Working with Resource Bundles"](#).

4.7 Working with Resource Bundles

When you define translatable strings (such as valedictory error messages, or attribute control hints for an entity object or view object), by default JDeveloper creates a project-level resource bundle file in which to store them. For example, when you define control hints for an entity object in the `StoreFront` project, JDeveloper creates the message bundle file named `StoreFrontBundle.xxx` for the package. The hints that you define can be used by generated forms and tables in associated view clients.

The resource bundle option that JDeveloper uses is determined by an option on the Resource Bundle page of the Project Properties dialog. By default JDeveloper sets the option to **Properties Bundle**, which produces a `.properties` file. For more information on this and other resource bundle options, see [Section 4.7.1, "How to Set Message Bundle Options"](#).

You can inspect the message bundle file for the entity object by selecting the object in the Application Navigator and looking in the corresponding **Sources** node in the

Structure window. The Structure window shows the implementation files for the component you select in the Application Navigator.

Example 4–3 shows a sample message bundle file where the control hint information appears. The first entry in each String array is a message key; the second entry is the locale-specific String value corresponding to that key.

Example 4–3 Project Message Bundle Stores Locale-Sensitive Control Hints

```
AddressUsageEO_OwnerTypeCode_Error_0=Invalid OwnerTypeCode.  
AddressUsageEO_UsageTypeCode_Error_0=Invalid UsageTypeCode.  
OwnerTypeCode_CONTROLTYPE=105  
PaymentOptionEO_RoutingIdentifier_Error_0=Please enter a valid routing identifier.  
PaymentOptionsEO_PaymentTypeCode_Error_0=Invalid PaymentTypeCode.  
PaymentTypeCode_CONTROLTYPE=105  
PaymentOption_AccountNumber=Please enter a valid Account Number  
MinPrice_FMT_FORMATTER=oracle.jbo.format.DefaultCurrencyFormatter  
CostPrice_FMT_FORMATTER=oracle.jbo.format.DefaultCurrencyFormatter  
UnitPrice_FMT_FORMATTER=oracle.jbo.format.DefaultCurrencyFormatter  
OrderEO_GiftMessage=Please supply a message shorter than 200 characters  
OrderEO=Please supply a gift message  
DiscountBaseEO_DiscountAmount=Discount must be between 0 and 40%  
  
oracle.fodemo.storefront.entities.PaymentOptionEO.ExpireDate_FMT_FORMAT=mm/yy  
#Date range validation for ValidFrom and ValidTo dates  
PaymentOptionEO_invalidDateRange_Error_0=Date range is invalid. {0} must be  
greater than {1}.  
PaymentOptionEO_DateRange_Error_0=Invalid date range.{0} should be greater than  
{1}.  
  
oracle.fodemo.storefront.entities.PaymentOptionEO.ValidFromDate_LABEL=Valid From  
Date  
oracle.fodemo.storefront.entities.PaymentOptionEO.ValidToDate_LABEL=Valid To Date  
OrderItemsVO_ImageId_Rule_0=ImageId not found  
oracle.fodemo.storefront.store.queries.AddressesVO.Address1_LABEL=Address  
oracle.fodemo.storefront.store.queries.AddressesVO.PostalCode_LABEL=Post Code or  
ZIP  
... .
```

4.7.1 How to Set Message Bundle Options

The resource bundle option JDeveloper uses to save control hints and other translatable strings is determined by an option on the Resource Bundle page of the Project Properties dialog. By default JDeveloper sets the option to **Properties Bundle** which produces a .properties file.

To set resource bundle options for your project

1. With your application open in JDeveloper, select a project and choose **Project Properties** from the **File** menu.
2. Click **Resource Bundle**.
3. Select the whether to use project or custom settings.

If you select **Use Custom Settings**, the settings apply only to your work with the current project. They are preserved between sessions, but are not recorded with the project and cannot be shared with other users. If you select **Use Project Settings**, your choices are recorded with the project and can be shared with others who use the project.

4. Specify your preference with the following options by selecting or deselecting the option:
 - **Automatically Synchronize Bundle**
 - **Warn About Hard-coded Translatable Strings**
 - **Always Prompt for Description**

For more information on these options, click **Help** to see the online help.

5. Select your choice of resource bundle granularity.
 - **One Bundle Per Project** (default)
 - **One Bundle Per File**
 - **Multiple Shared Bundles** (not available for ADF Business Components)
6. Select the type of file to use.
 - **List Resource Bundle**
The ListResourceBundle class manages resources in a name/value array. Each ListResourceBundle class is contained within a Java class file. You can store any locale-specific object in a ListResourceBundle class.
 - **Properties Bundle** (default)
A text file containing translatable text in name/value pairs. Property files (like the one shown in [Example 4–3](#)) can contain values only for String objects. If you need to store other types of objects, you must use a ListResourceBundle instead.
 - **Xliff Resource Bundle**
The XML Localization Interchange File Format (XLIFF) is an XML-based format for exchanging localization data.
7. Click **OK** to apply your settings and close the dialog.

4.7.2 How to Use Multiple Resource Bundles

When you define translatable strings (for example, for attribute control hints), the Select Text Resource dialog allows you to enter a new string or select one that is already defined in the default resource bundle for the object. You can also use a different resource bundle if necessary. This is helpful when you use a common resource bundle that is shared between projects.

To use strings in a non-default resource bundle:

1. In the Select Text Resource dialog, select the bundle you want to use from the **Resource Bundle** dropdown list.

If the desired resource bundle is not included in the **Resource Bundle** dropdown list, click the **Browse** icon to locate and select the resource bundle you want to use.

The dialog displays the strings that are currently defined in the selected resource bundle.

2. Select an existing string and click **Select**, or enter a new string and click **Save and Select**.

If you entered a new string it is written to the selected resource bundle.

4.7.3 How to Internationalize the Date Format

Internationalizing the model layer of an application built using ADF Business Components entails producing translated versions of each component message bundle file. For example, the Italian version of the OrdersImplMsgBundle message bundle would be a class named OrdersImplMsgBundle_it and a more specific Swiss Italian version would have the name OrdersImplMsgBundle_it_ch. These classes typically extend the base message bundle class, and contain entries for the message keys that need to be localized, together with their localized translation.

[Example 4-4](#) shows the Italian version of an entity object message bundle. Notice that in the Italian translation, the format masks for RequestDate and AssignedDate have been changed to dd/MM/yyyy HH:mm. This ensures that an Italian user will see a date value like May 3rd, 2006, as 03/05/2006 15:55, instead of 05/03/2006 15:55, which the format mask in the default message bundle would produce. Notice the overridden getContents() method. It returns an array of messages with the more *specific* translated strings merged together with those that are not overridden from the superclass bundle. At runtime, the appropriate message bundles are used automatically, based on the current user's locale settings.

Example 4-4 Localized Entity Object Component Message Bundle for Italian

```
package devguide.model.entities.common;
import oracle.jbo.common.JboResourceBundle;
public class ServiceRequestImplMsgBundle_it
    extends ServiceRequestImplMsgBundle {
    static final Object[][] sMessageStrings = {
        { "AssignedDate_FMT_FORMAT", "dd/MM/yyyy HH:mm" },
        { "AssignedDate_LABEL", "Assegnato il" },
        { "AssignedTo_LABEL", "Assegnato a" },
        { "CreatedBy_LABEL", "Aperto da" },
        { "ProblemDescription_LABEL", "Problema" },
        { "RequestDate_FMT_FORMAT", "dd/MM/yyyy HH:mm" },
        { "RequestDate_LABEL", "Aperto il" },
        { "RequestDate_TOOLTIP", "La data in cui il ticket è stato aperto" },
        { "Status_LABEL", "Stato" },
        { "SrvId_LABEL", "Ticket" }
    };
    public Object[][] getContents() {
        return super.getMergedArray(sMessageStrings, super.getContents());
    }
}
```

4.8 Defining Business Logic Groups

Business logic groups allow you to encapsulate a set of related control hints, default values, and validation logic. A business logic group is maintained separate from the base entity in its own file, and can be enabled dynamically based on context values of the current row.

This is useful, for example, for an HR application that defines many locale-specific validations (like national identifier or tax law checks) that are maintained by a dedicated team for each locale. The business logic group eases maintenance by storing these validations in separate files, and optimizes performance by loading them only when they are needed.

Each business logic group contains a set of business logic units. Each unit identifies the set of business logic that is loaded for the entity, based on the value of the attribute associated with the business logic group.

For example, you can define a business logic group for an Employee entity object, specifying the EmpRegion attribute as the discriminator. Then define a business logic unit for each region, one that specifies a range validator for the employee's salary. When the application loads a row from the Employee entity, the appropriate validator for the EmpSalary attribute is loaded (based on the value of the EmpRegion attribute).

4.8.1 How to Create a Business Logic Group

You create the business logic group for an entity object from the overview editor.

To create a business logic group:

1. In the Application Navigator, double-click the entity for which you want to create a business logic group to open it in the overview editor.
2. In the **Business Logic Groups** section (on the **General** tab), click the **Add** icon.
3. In the creation dialog, select the appropriate group discriminator attribute and specify a name for the group.

Tip: To enhance the readability of your code, you can name the group to reflect the discriminator. For example, if the group discriminator attribute is EmpRegion, you can name the business logic group BlgRegion.

4. Click **OK**.

The new business logic group is added to the table in the overview editor. After you have created the group, you can add business logic units to it.

4.8.2 How to Create a Business Logic Unit

You can create a business logic unit from the New Gallery, or directly from the context menu of the entity that contains the business logic group.

To create a business logic unit:

1. Select the project in the Application Navigator and choose **New** from the **File** menu. Then in the New Gallery, expand **Business Tier**, select **ADF Business Components**, then select **Entity Business Logic Unit** and click **OK**.

Alternatively, you can right-click the entity that contains the business logic group and choose **New Entity Business Logic Unit** from the context menu.

2. In the Create Business Logic Unit dialog, specify the name of the base entity and select the appropriate business logic group.
3. Enter a name for the business logic unit.

The name of each business logic unit must reflect a valid value of the group discriminator attribute with which this business logic unit will be associated. For example, if the group discriminator attribute is EmpRegion, the name of the business logic unit associated with the EmpRegion value of US must be US.

4. Specify the package for the business logic unit.

Note: The package for the business logic unit does not need to be the same as the package for the base entity or the business logic group. This allows you to develop and deliver business logic units separately from the core application.

5. Click **OK**.

JDeveloper creates the business logic unit and opens it in the overview editor. After you have created the unit, you can redefine the business logic for it.

4.8.3 How to Override Attributes in a Business Logic Unit

When you view the Attributes page for the business logic unit (in the overview editor), you can see that the **Extends** column in the attributes table shows that the attributes are "extended" in the business logic unit. If you edit an extended attribute, your changes are implemented in the base entity. To implement these changes in the business logic unit rather than the base entity, you must define attributes as overridden in the business logic unit before you edit them.

To override attributes in a business logic unit:

1. In the Application Navigator, double-click the business logic unit to open it in the overview editor.
2. On the Attributes page, click the **Override** button.
3. In the Inherited Attributes to Override dialog, specify the attributes you want to override and click **OK**.

After you make an attribute overridden, you can edit the attribute as you normally would by double-clicking the attribute to open it in the Edit Attribute dialog. You will notice that in an overridden attribute, you are limited to making modifications to only control hints, validators, and default values.

4.8.4 What Happens When You Create a Business Logic Group

When you create a business logic group, JDeveloper adds a reference to the group in the base entity's XML file. [Example 4–5](#) shows the code added to the base entity's XML file for the business logic group.

Example 4–5 XML Code in the Base Entity for a Business Logic Group

```
<BusLogicGroup
    Name="BlgRegion"
    DiscrAttrName="EmpRegion"/>
```

When you create a business logic unit, JDeveloper generates an XML file similar to that of an entity object. [Example 4–6](#) shows XML code for a business logic unit.

Note: The package for the business logic unit does not need to be the same as the package for the base entity or the business logic group. This allows you to develop and deliver business logic units separately from the core application.

Example 4–6 XML Code for a Business Logic Unit

```
<Entity
```

```

xmlns="http://xmlns.oracle.com/bc4j"
Name="Emp_BlgRegion_US"
Extends="myApplication.common.Emp"
. . .
BusLogicGroupName="BlgRegion"
BusLogicUnitName="US"
xmlns:validation="http://xmlns.oracle.com/adfm/validation">
<validation:RangeValidationBean
    Name="Emp_BlgRegion_US_Rule_0"
    ResId="myApplication.common.Emp_BlgRegion_US_Rule_0"
    OnAttribute="EmpSal"
    OperandType="LITERAL"
    Inverse="false"
    MinValue="1000"
    MaxValue="9000">
    <validation:OnAttributes>
        <validation:Item
            Value="EmpSal"/>
    </validation:OnAttributes>
    <validation:ResExpressions>
        <validation:Expression
            Name="0"><! [CDATA[min]]></validation:Expression>
        <validation:Expression
            Name="1"><! [CDATA[max]]></validation:Expression>
    </validation:ResExpressions>
</validation:RangeValidationBean>
<ResourceBundle>
    <PropertiesBundle
        PropertiesFile="myApplication.common.commonBundle"/>
</ResourceBundle>
</Entity>

```

4.8.5 What Happens at Runtime: Invoking a Business Logic Group

When a row is loaded in the application at runtime, the entity object decides which business logic units to apply to it.

The base entity maintains a list of business logic groups. Each group references the value of an attribute on the entity, and this value determines which business logic unit to load for that group. This evaluation is performed for each row that is loaded.

If the logic for determining which business logic unit to load is more complex than just a simple attribute value, you can create a transient attribute on the entity object, and use a groovy expression to determine the value of the transient attribute.

4.9 Configuring Runtime Behavior Declaratively

Entity objects offer numerous declarative features to simplify implementing typical enterprise business applications. Depending on the task, sometimes the declarative facilities *alone* may satisfy your needs. The declarative runtime features that describe the basic persistence features of an entity object are covered in this section, while declarative validation and business rules are covered in [Chapter 7, "Defining Validation and Business Rules Declaratively"](#)

Note: It is possible to go beyond the declarative behavior to implement more complex business logic or validation rules for your business domain layer when needed. In [Chapter 8, "Implementing Validation and Business Rules Programmatically"](#), you'll see some of the most typical ways that you extend entity objects with custom code.

Also, it is important to note as you develop your application that the business logic you implement, either programmatically or declaratively, should not assume that the attributes of an entity object or view row will be set in a particular order. This will cause problems if the end user enters values for the attributes in an order other than the assumed one.

4.9.1 How to Configure Declarative Runtime Behavior

To configure the declarative runtime behavior of an entity object, use the overview editor.

To configure the declarative runtime behavior of an entity object:

1. In the Application Navigator, double-click the entity object to open the editor.
2. In the overview editor, click the **General** tab to view the name and package of the entity object, and configure aspects of the object at the entity level, such as its associated schema, validation rules, custom properties, and security.
 - The **Entity Validation Rules** node allows you to declare validation rules, which can optionally use Groovy scripts. For information on how to use the declarative validation features, see [Chapter 7, "Defining Validation and Business Rules Declaratively"](#).
 - The **Tuning** node allows you to set options to make database operations more efficient when you create, modify, or delete multiple entities of the same type in a single transaction. For more information, see [Section 34.3, "Using Update Batching"](#).
 - The **Custom Properties** node allows you to define custom metadata that you can access at runtime on the entity.
 - The **Security** node allows you to define role-based updatability permissions for the entity. For more information, see [Chapter 28, "Adding Security to a Fusion Web Application"](#)
3. Click the **Attributes** tab to create or delete attributes that represent the data relevant to an entity object, and configure aspects of the attribute, such as validation rules, custom properties, and security.

Select an attribute and click the **Edit** icon to access the properties of the attribute. For information on how to set these properties, see [Section 4.10, "Setting Attribute Properties"](#).

Note: If your entity has a long list of attribute names, there's a quick way to find the one you're looking for. In the Structure window with the **Attributes** node expanded, you can begin to type the letters of the attribute name and JDeveloper performs an incremental search to take you to its name in the tree.

4. Click the **Java** tab to select the classes you generate for custom Java implementation. You can use the Java classes for such things as defining programmatic business rules, as in [Chapter 8, "Implementing Validation and Business Rules Programmatically"](#).
5. Click the **View Accessors** tab to create and manage view accessors. For more information, see [Section 10.4.1, "How to Create a View Accessor for an Entity Object or View Object"](#).

4.9.2 What Happens When You Configure Declarative Runtime Behavior

The declarative settings that describe and control an entity object's runtime behavior are stored in its XML component definition file. When you use the overview editor to modify settings of your entity, JDeveloper updates the component's XML definition file and optional custom Java files.

4.10 Setting Attribute Properties

The declarative framework helps you set attribute properties easily. In all cases, you set these properties in the Edit Attribute dialog, which you can access from the Attributes page of the overview editor.

4.10.1 How to Set Database and Java Data Types for an Entity Object Attribute

The **Persistent** property controls whether the attribute value corresponds to a column in the underlying table, or whether it is just a transient value. If the attribute is persistent, the **Database Column** area lets you change the name of the underlying column that corresponds to the attribute and indicate its column type with precision and scale information (e.g. VARCHAR2 (40) or NUMBER (4, 2)). Based on this information, at runtime the entity object enforces the maximum length and precision/scale of the attribute value, and throws an exception if a value does not meet the requirements.

Both the Business Components from Tables wizard and the Create Entity Object wizard infer the Java type of each entity object attribute from the SQL type of the database column type of the column to which it is related.

Note: The project's Type Map setting also plays a role in determining the Java data type. You specify the Type Map setting when you initialize your business components project, before any business components are created. For more information, see [Section 3.3.1, "Choosing a Connection, SQL Flavor, and Type Map"](#).

The **Attribute Type** field (in the Edit Attribute dialog) allows you to change the Java type of the entity attribute to any type you might need. The **Database Column Type** field reflects the SQL type of the underlying database column to which the attribute is mapped. The value of the **Database Column Name** field controls the column to which the attribute is mapped.

Your entity object can handle tables with various column types, as listed in [Table 4–1](#). With the exception of the `java.lang.String` class, the default Java attribute types are all in the `oracle.jbo.domain` and `oracle.ord.im` packages and support efficiently working with Oracle database data of the corresponding type. The dropdown list for the **Attribute Type** field includes a number of other common Java types that are also supported.

Table 4–1 Default Entity Object Attribute Type Mappings

Oracle Column Type	Entity Column Type	Entity Java Type
NVARCHAR2 (n), VARCHAR2 (n) , NCHAR VARYING (n) , VARCHAR (n)	VARCHAR2	java.lang.String
NUMBER	NUMBER	oracle.jbo.domain.Number
DATE	DATE	oracle.jbo.domain.Date
TIMESTAMP(n), TIMESTAMP(n) WITH TIME ZONE, TIMESTAMP(n) WITH LOCAL TIME ZONE	TIMESTAMP	java.sql.Timestamp
LONG	LONG	java.lang.String
RAW(n)	RAW	oracle.jbo.domain.Raw
LONG RAW	LONG RAW	oracle.jbo.domain.Raw
ROWID	ROWID	oracle.jbo.domain.RowID
NCHAR, CHAR	CHAR	oracle.jbo.domain.Char
CLOB	CLOB	oracle.jbo.domain.ClobDomain
NCLOB	NCLOB	oracle.jbo.domain.NClobDomain
BLOB	BLOB	oracle.jbo.domain.BlobDomain
BFILE	BFILE	oracle.jbo.domain.BFileDomain
ORDSYS.ORDIMAGE	ORDSYS.ORDIMAGE	oracle.ord.im.OrdImageDomain
ORDSYS.ORDVIDEO	ORDSYS.ORDVIDEO	oracle.ord.im.OrdVideoDomain
ORDSYS.ORDAUDIO	ORDSYS.ORDAUDIO	oracle.ord.im.OrdAudioDomain
ORDSYS.ORDDOC	ORDSYS.ORDDOC	oracle.ord.im.OrdDocDomain

Note: In addition to the types mentioned here, you can use any Java object type as an entity object attribute's type, provided it implements the `java.io.Serializable` interface.

4.10.2 How to Indicate Data Type Length, Precision, and Scale

When working with types that support defining a maximum length like `VARCHAR2 (n)`, the **Database Column Type** field includes the maximum attribute length as part of the value. For example, an attribute based on a `VARCHAR2 (10)` column in the database will initially reflect the maximum length of 10 characters by showing `VARCHAR2 (10)` as the database column type. If for some reason you want to restrict the maximum length of the `String`-valued attribute to fewer characters than the underlying column will allow, just change the maximum length of the **Database Column Type** value.

For example, if the `EMAIL` column in the `PERSONS` table is `VARCHAR2 (50)`, then by default the `Email` attribute in the `Persons` entity object defaults to the same. But if you know that the actual email addresses are always 8 characters or fewer, you can update the database column type for the `Email` attribute to be `VARCHAR2 (8)` to enforce a maximum length of 8 characters at the entity object level.

The same holds for attributes related to database column types that support defining a precision and scale like `NUMBER (p [, s])`. For example, to restrict an attribute based

on a NUMBER (7, 2) column in the database to instead have a precision of 5 and a scale of 1, just update the value of the **Database Column Type** field to be NUMBER (5, 1).

4.10.3 How to Control the Updatability of an Attribute

The **Updatable** property controls when the value of a given attribute can be updated. You can select the following values:

- **Always**, the attribute is always updatable
- **Never**, the attribute is read-only
- **While New**, the attribute can be set during the transaction that creates the entity row for the first time, but after being successfully committed to the database the attribute is read-only

Note: In addition to the static declaration of updatability, you can also add custom code in the `isAttributeUpdateable()` method of the entity to determine the updatability of an attribute at runtime.

4.10.4 How to Make an Attribute Mandatory

Select the **Mandatory** checkbox if the field is required. The mandatory property is enforced during entity-level validation at runtime (and not when the attribute validators are run).

4.10.5 How to Define the Primary Key for the Entity

The **Primary Key** property indicates whether the attribute is part of the key that uniquely identifies the entity. Typically, you use a single attribute for the primary key, but multiattribute primary keys are fully supported.

At runtime, when you access the related **Key** object for any entity row using the `getKey()` method, this **Key** object contains the value of the primary key attribute for the entity object. If your entity object has multiple primary key attributes, the **Key** object contains each of their values. It is important to understand that these values appear in the same *relative* sequential order as the corresponding primary key attributes in the entity object definition.

For example, if the `OrderItemEO` entity object has multiple primary key attributes `OrderId` and `LineItemId`. On the Entity Attribute page of the overview editor, `OrderId` is first, and `LineItemId` is second. An array of values encapsulated by the **Key** object for an entity row of type `OrderItemEO` will have these two attribute values in exactly this order.

It is crucial to be aware of the order in which multiple primary key attributes appear on the Entity Attributes page. If you try to use `findByPrimaryKey()` to find an entity with a multiattribute primary key, and the **Key** object you construct has these multiple primary key attributes in the wrong order, the entity row will not be found as expected.

4.10.6 How to Define a Static Default Value

The **Value** field in the Edit Attribute dialog allows you to specify a static default value for the attribute when the **Value Type** is set to **Literal**. For example, you can set the default value of the `ServiceRequest` entity object's `Status` attribute to `Open`, or set the default value of the `User` entity object's `UserRole` attribute to `user`.

Note: When more than one attribute is defaulted for an entity object, the attributes are defaulted in the order in which they appear in the entity object's XML file.

4.10.7 How to Define a Default Value Using a Groovy Expression

You can use a Groovy expression to define a default value for an attribute. This approach is useful if you want to be able to change default values at runtime, but if the default value is always the same, the value is easier to see and maintain using the Default field. For general information about using Groovy, see [Section 3.6, "Overview of Groovy Support"](#).

To define a default value using a Groovy expression:

1. In the Application Navigator, double-click the entity to open the overview editor.
2. In the overview editor, click the **Attributes** tab.
3. Select an attribute and click the Edit icon.
4. In the Edit Attribute, select **Expression** for the value type, and click **Edit** (next to the **Value** field).
5. Enter a Groovy expression in the field provided, and click **OK**.
6. Click **OK**.

4.10.8 What Happens When You Create a Default Value Using a Groovy expression

When you define a default value using a Groovy expression, a `<TransientExpression>` tag is added to the entity object's XML file within the appropriate attribute. [Figure 4-7](#) shows sample XML code for an Groovy expression that gets the current date for a default value.

Example 4-7 Default Date Value

```
<TransientExpression>
  <![CDATA[
    newValue <= adf.currentDate
  ]]>
</TransientExpression>
```

4.10.9 How to Synchronize with Trigger-Assigned Values

If you know that the underlying column value will be updated by a database trigger during insert or update operations, you can enable the respective **Insert** or **Update** checkboxes in the **Refresh After** area to ensure the framework automatically retrieves the modified value and keeps the entity object and database row in sync. The entity object will use the Oracle SQL `RETURNING INTO` feature, while performing the `INSERT` or `UPDATE` to return the modified column back to your application in a single database roundtrip.

Note: If you create an entity object for a synonym that resolves to a remote table over a DBLINK, use of this feature will give an error at runtime like:

```
JBO-26041: Failed to post data to database during "Update"
## Detail 0 ##
ORA-22816: unsupported feature with RETURNING clause
```

[Section 34.6, "Basing an Entity Object on a Join View or Remote DBLink"](#) describes a technique to circumvent this database limitation.

4.10.10 How to Get Trigger-Assigned Primary Key Values from a Database Sequence

One common case for refreshing an attribute after insert occurs when a primary key attribute value is assigned by a BEFORE INSERT FOR EACH ROW trigger. Often the trigger assigns the primary key from a database sequence using PL/SQL logic. [Example 4-8](#) shows an example of this.

Example 4-8 PL/SQL Code Assigning a Primary Key from a Database Sequence

```
CREATE OR REPLACE TRIGGER ASSIGN_SVR_ID
BEFORE INSERT ON SERVICE_REQUESTS FOR EACH ROW
BEGIN
IF :NEW.SVR_ID IS NULL OR :NEW.SVR_ID < 0 THEN
SELECT SERVICE_REQUESTS_SEQ.NEXTVAL
INTO :NEW.SVR_ID
FROM DUAL;
END IF;
END;
```

Set the value of the **Type** field to the built-in data type named DBSequence and the primary key will be assigned automatically by the database sequence. Setting this data type automatically selects the refresh after **Insert** checkbox.

Note: The sequence name shown on the **Sequence** tab is used only at design time when you use the **Create Database Tables** feature described in [Section 4.2.6, "How to Create Database Tables from Entity Objects"](#). The sequence indicated here will be created along with the table on which the entity object is based.

When you create a new entity row whose primary key is a 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. If you are creating a set of interrelated entities in the same transaction, you can assign this temporary value as a foreign key value on other new, related entity rows. At transaction commit time, the entity object issues its `INSERT` operation using the `RETURNING INTO` clause to retrieve the actual database trigger-assigned primary key value. In a composition relationship, any related new entities that previously used the temporary negative value as a foreign key will get that value updated to reflect the actual new primary key of the master.

You will typically also set the **Updatable** property of a DBSequence-valued primary key to **Never**. The entity object assigns the temporary ID, and then refreshes it with the actual ID value after the `INSERT` operation. The end user never needs to update this value.

For information on how to implement this functionality for an association that is not a composition, see [Section 34.8.3.3, "Understanding Associations Based on DBSequence-Valued Primary Keys"](#).

Note: For a metadata-driven alternative to the DBSequence approach, see [Section 4.11.5, "Assigning the Primary Key Value Using an Oracle Sequence"](#).

4.10.11 How to Protect Against Losing Simultaneously Updated Data

At runtime the framework provides automatic "lost update" detection for entity objects to ensure that a user cannot unknowingly modify data that another user has updated and committed in the meantime. Typically, this check is performed by comparing the original values of each persistent entity attribute against the corresponding current column values in the database at the time the underlying row is locked. Before updating a row, the entity object verifies that the row to be updated is still consistent with the current state of the database. If the row and database state are inconsistent, then the entity object raises the `RowInconsistentException`.

You can make the lost update detection more efficient by identifying any attributes of your entity whose values you know will be updated whenever the entity is modified. Typical candidates include a version number column or an updated date column in the row. The change-indicator attribute value might be assigned by a database trigger you've written and refreshed in the entity object using the **Refresh After Insert** and **Refresh After Update** options. Alternatively, you can indicate that the entity object should manage updating the change-indicator attribute's value using the history attribute feature described in [Section 4.10.12, "How to Track Created and Modified Dates Using the History Column"](#). To detect whether the row has been modified since the user queried it in the most efficient way, select the **Change Indicator** option to compare only the change-indicator attribute values.

4.10.12 How to Track Created and Modified Dates Using the History Column

If you need to keep track of historical information in your entity object, such as when an entity was created or modified and by whom, or the number of times the entity has been modified, you specify an attribute with the **History Column** option selected.

If an attribute's data type is `Number`, `String`, or `Date`, and if it is not part of the primary key, then you can enable this property to have your entity automatically maintain the attribute's value for historical auditing. How the framework handles the attribute depends which type of history attribute you indicate:

- **Created On** - This attribute is populated with the time stamp of when the row was created. The time stamp is obtained from the database.
- **Created By** - The attribute is populated with the name of the user who created the row. The user name is obtained using the `getUserPrincipleName()` method on the `Session` object.
- **Modified On** - This attribute is populated with the time stamp whenever the row is updated/created.
- **Modified By** - This attribute is populated with the name of the user who creates or updates the row.
- **Version Number** - This attribute is populated with a long value that is incremented whenever a row is created or updated.

4.10.13 How to Configure Composition Behavior

An entity object exhibits composition behavior when it creates (or composes) other entities, such as an `OrderEO` entity creating a `OrderItemEO` entity. This additional runtime behavior determines its role as a logical container of other nested entity object parts.

Note: Composition also affects the order in which entities are validated. For more information, see [Section 7.2.3, "Understanding the Impact of Composition on Validation Order"](#).

The features that are always enabled for composing entity objects are described in the following sections:

- [Section 4.10.13.1, "Orphan-Row Protection for New Composed Entities"](#)
- [Section 4.10.13.2, "Ordering of Changes Saved to the Database"](#)
- [Section 4.10.13.3, "Cascade Update of Composed Details from Refresh-On-Insert Primary Keys"](#)

The additional features, and the properties that affect their behavior, are described in the following sections:

- [Section 4.10.13.4, "Cascade Delete Support"](#)
- [Section 4.10.13.5, "Cascade Update of Foreign Key Attributes When Primary Key Changes"](#)
- [Section 4.10.13.6, "Locking of Composite Parent Entities"](#)
- [Section 4.10.13.7, "Updating of Composing Parent History Attributes"](#)

4.10.13.1 Orphan-Row Protection for New Composed Entities

When a composed entity object is created, it performs an existence check on the value of its foreign key attribute to ensure that it identifies an existing entity as its owning parent entity. At create time, if no foreign key is found or else a value that does not identify an existing entity object is found, the entity object throws an `InvalidOwnerException` instead of allowing an orphaned child row to be created without a well-identified parent entity.

Note: The existence check finds new pending entities in the current transaction, as well as existing ones in the database if necessary.

4.10.13.2 Ordering of Changes Saved to the Database

Composition behavior ensures that the sequence of data manipulation language (DML) operations performed in a transaction involving both composing and composed entity objects is performed in the correct order. For example, an `INSERT` statement for a new composing parent entity object will be performed before the DML operations related to any composed children.

4.10.13.3 Cascade Update of Composed Details from Refresh-On-Insert Primary Keys

When a new entity row having a primary key configured to refresh on insert is saved, then after its trigger-assigned primary value is retrieved, any composed entities will have their foreign key attribute values updated to reflect the new primary key value.

There are a number of additional composition related features that you can control through settings on the Association Properties page of the Create Association wizard or the overview editor. [Figure 4–12](#) shows the Relationships page for the OrderItemsOrdersFkAssoc association between two entity objects: OrderItemEO and OrderEO.

4.10.13.4 Cascade Delete Support

You can either enable or prevent the deletion of a composing parent while composed children entities exist. When the **Implement Cascade Delete** option (see [Figure 4–12](#)) is deselected, the removal of the composing entity object is prevented if it contains any composed children.

Figure 4–12 Composition Settings on Relationship Page of Overview Editor for Associations



When selected, this option allows the composing entity object to be removed unconditionally together with any composed children entities. If the related **Optimize for Database Cascade Delete** option is deselected, then the composed entity objects perform their normal DELETE statement at transaction commit time to make the changes permanent. If the option is selected, then the composed entities do not perform the DELETE statement on the assumption that the database ON DELETE CASCADE constraint will handle the deletion of the corresponding rows.

4.10.13.5 Cascade Update of Foreign Key Attributes When Primary Key Changes

Select the **Cascade Update Key Attributes** option (see [Figure 4–12](#)) to enable the automatic update of the foreign key attribute values in composed entities when the primary key value of the composing entity is changed.

4.10.13.6 Locking of Composite Parent Entities

Select the **Lock Top-Level Container** option (see [Figure 4–12](#)) to control whether adding, removing, or modifying a composed detail entity row should attempt to lock the composing entity before allowing the changes to be saved.

4.10.13.7 Updating of Composing Parent History Attributes

Select the **Update Top-Level History Columns** option (see [Figure 4–12](#)) to control whether adding, removing, or modifying a composed detail entity object should update the **Modified By** and **Modified On** history attributes of the composing parent entity.

4.10.14 How to Set the Discriminator Attribute for Entity Object Inheritance Hierarchies

Sometimes a single database table stores information about several different kinds of logically related objects. For example, a payroll application might work with hourly, salaried, and contract employees all stored in a single EMPLOYEES table with an

`EMPLOYEE_TYPE` column. In this case, the value of the `EMPLOYEE_TYPE` column contains values like H, S, or C to indicate respectively whether a given row represents an hourly, salaried, or contract employee. And while it is possible that many attributes and behavior are the same for all employees, certain properties and business logic may also depend on the type of employee.

In situations where common information exists across related objects, it may be convenient to represent these different types of entity objects using an inheritance hierarchy. For example, attributes and methods common to all employees can be part of a base `Employee` entity object, while subtype entity objects like `HourlyEmployee`, `SalariedEmployee`, and `ContractEmployee` extend the base `Employee` object and add additional properties and behavior. The **Discriminator** attribute setting is used to indicate which attribute's value distinguishes the type of row. [Section 34.7, "Using Inheritance in Your Business Domain Layer"](#) explains how to set up and use inheritance.

4.10.15 How to Define Alternate Key Values

Database primary keys are often generated from a sequence and may not be data you want to expose to the user for a variety of reasons. For this reason, it's often helpful to have alternate key values that are unique. For example, you might want to enforce that every customer have a unique email address. Because a customer may change their email address, you won't want to use that value as a primary key, but you still want the user to have a unique field they can use for login or other purposes.

Alternate keys are useful for direct row lookups via the `findByPrimaryKey` class of methods. Alternate keys are frequently used for efficient uniqueness checks in the middle tier. For information on how to find out if a value is unique, see [Section 7.4.1, "How to Ensure That Key Values Are Unique"](#).

To define an alternate key, you use the Create Entity Constraint wizard.

To define alternate key values:

1. In the Application Navigator, right-click an entity object and choose **New Entity Constraint**.
2. Follow the steps in the Create Entity Constraint wizard to name your constraint and select the attribute or attributes that participate in the key.
3. On the Properties page, select **Alternate Key** and choose the appropriate **Key Properties** options.

For more information about the **Key Properties** options, press the F1 key or click **Help**.

4.10.16 What Happens When You Define Alternate Key Values

When you define alternate key values, a hashmap is created for fast access to entities that are already in memory.

4.10.17 What You May Need to Know About Alternate Key Values

The Unique key constraint is used only for forward generation of `UNIQUE` constraints in the database, not for alternate key values.

4.11 Working Programmatically with Entity Objects and Associations

You may not always need or want UI-based or programmatic clients to work directly with entity objects. Sometimes, you may just want to use an external client program to access an application module and work directly with the view objects in its data model. [Chapter 5, "Defining SQL Queries Using View Objects"](#) describes how to easily combine the flexible SQL-querying of view objects with the business logic enforcement and automatic database interaction of entity objects to build powerful applications. The combination enables a fully updatable application module data model, designed to meet the needs of the current end-user tasks at hand, that shares the centralized business logic in your reusable domain business object layer.

However, it is important first to understand how view objects and entity objects can be used on their own before learning to harness their combined power. By learning about these objects in greater detail, you will have a better understanding of when you should use them alone and when to combine them in your own applications.

Since clients don't work *directly* with entity objects, any code you write that works programmatically with entity objects will typically be custom code in a custom application module class or in the custom class of another entity object.

4.11.1 How to Find an Entity Object by Primary Key

To access an entity row, you use a related object called the *entity definition*. At runtime, each entity object has a corresponding entity definition object that describes the structure of the entity and manages the instances of the entity object it describes. After creating an application module and enabling a custom Java class for it, imagine you wanted to write a method to return a specific order. It might look like the `retrieveOrderById()` method shown in [Example 4–9](#).

To find an entity object by primary key:

1. Find the entity definition.

You obtain the entity definition object for the `OrderEO` entity by passing its fully qualified name to the static `getDefinitionObject()` method imported from the `EntityDefImpl` class. The `EntityDefImpl` class in the `oracle.jbo.server` package implements the entity definition for each entity object.

2. Construct a key.

You build a `Key` object containing the primary key attribute that you want to look up. In this case, you're creating a key containing the single `orderId` value passed into the method as an argument.

3. Find the entity object using the key.

You use the entity definition's `findByPrimaryKey()` method to find the entity object by key, passing in the current transaction object, which you can obtain from the application module using its `getDBTransaction()` method. The concrete class that represents an entity object row is the `oracle.jbo.server.EntityImpl` class.

4. Return the object or some of its data to the caller.

[Example 4–9](#) show example code for a `retrieveOrderById()` method developed using this basic procedure.

Example 4–9 Retrieving an OrderEO Entity Object by Key

```
/* Helper method to return an Order by Id */
private OrderEOImpl retrieveOrderById(long orderId) {
    EntityDefImpl orderDef = OrderEOImpl.getDefinitionObject();
    Key orderKey = OrderEOImpl.createPrimaryKey(new DBSequence(orderId));
    return (OrderEOImpl)orderDef.findByPrimaryKey(getDBTransaction(), orderKey);
}
```

Note: The `oracle.jbo.Key` object constructor can also take an Object array to support creating multiattribute keys, in addition to the more typical single-attribute value keys.

4.11.2 How to Access an Associated Entity Using the Accessor Attribute

You can create a method to access an associated entity based on an accessor attribute that requires no SQL code. For example, the method `findOrderCustomer()` might find an order, then access the associated `PersonEO` entity object representing the customer assigned to the order. For an explanation of how associations enable easy access from one entity object to another, see [Section 4.3, "Creating and Configuring Associations"](#).

To avoid a conflict with an existing method in the application module that finds the same associated entity using the same accessor attribute, you can refactor this functionality into a helper method that you can then reuse anywhere in the application module it is required. For example, the `retrieveOrderById()` method (shown in [Example 4–9](#)) refactors the functionality that finds an order.

To access an associated entity object using the accessor attribute:

1. Find the associated entity by the accessor attribute.

The `findOrderCustomer()` method uses the `retrieveOrderById()` helper method to retrieve the `OrderEO` entity object by ID.

2. Access the associated entity using the accessor attribute.

Using the attribute getter method, you can pass in the name of an association accessor and get back the entity object on the other side of the relationship. (Note that [Section 4.3.3, "How to Change Entity Association Accessor Names"](#) explains that renaming the association accessor allows it to have a more intuitive name.)

3. Return some of its data to the caller.

The `findOrderCustomer()` method uses the getter methods on the returned `PersonEO` entity to return the assigned customer's name by concatenation their first and last names.

Notice that you did not need to write any SQL to access the related `PersonEO` entity. The relationship information captured in the ADF association between the `OrderEO` and `PersonEO` entity objects is enough to allow the common task of data navigation to be automated.

[Example 4–10](#) shows the code for `findOrderCustomer()` that uses the helper method.

Example 4–10 Accessing an Associated Entity Using the Accessor Attribute

```
/* Access an associated Customer entity from the Order entity */
public String findOrderCustomer(long orderId) {
    //1. Find the OrderEO object
```

```
OrderEOImpl order = retrieveOrderById(orderId);
if (order != null) {
    //2. Access the PersonEO object using the association accessor attribute
    PersonEOImpl cust = (PersonEOImpl)order.getPerson();
    if (cust != null) {
        //3. Return attribute values from the associated entity object
        return cust.getFirstName() + " " + cust.getLastName();
    }
    else {
        return "Unassigned";
    }
}
else {
    return null;
}
}
```

4.11.3 How to Update or Remove an Existing Entity Row

Once you've got an entity row in hand, it's simple to update it or remove it. You could add a method like the `updateOrderStatus()` shown in [Example 4-11](#) to handle the job.

To update an entity row:

1. Find the Order by ID.

Using the `retrieveOrderById()` helper method, the `updateOrderStatus()` method retrieves the `OrderEO` entity object by Id.

2. Set one or more attributes to new values.

Using the `EntityImpl` class' `setAttribute()` method, the `updateOrderStatus()` method updates the value of the `Status` attribute to the new value passed in.

3. Commit the transaction.

Using the application module's `getDBTransaction()` method, the `updateOrderStatus()` method accesses the current transaction object and calls its `commit()` method to commit the transaction.

Example 4-11 Updating an Existing Entity Row

```
/* Update the status of an existing order */
public void updateOrderStatus(long orderId, String newStatus) {
    //1. Find the order
    OrderEOImpl order = retrieveOrderById(orderId);
    if (order != null) {
        //2. Set its Status attribute to a new value
        order.setOrderStatusCode(newStatus);
        //3. Commit the transaction
        try {
            getDBTransaction().commit();
        }
        catch (JboException ex) {
            getDBTransaction().rollback();
            throw ex;
        }
    }
}
```

The example for *removing* an entity row would be the same, except that after finding the existing entity, you would use the following line instead to remove the entity before committing the transaction:

```
// Remove the entity instead!
order.remove();
```

4.11.4 How to Create a New Entity Row

In addition to using the entity definition to find existing entity rows, you can also use it to create new ones. In the case of product entities, you could write a `createProduct()` method like the one shown in [Example 4–12](#) to accept the name and description of a new product, and return the new product ID assigned to it. This example assumes that the `ProductId` attribute of the `ProductBaseEO` entity object has been updated to have the `DBSequence` type (see [Section 4.10.10, "How to Get Trigger-Assigned Primary Key Values from a Database Sequence"](#)). This setting ensures that the attribute value is refreshed to reflect the value of the trigger from the corresponding database table, assigned to it from the table's sequence in the application schema.

To create an entity row:

1. Find the entity definition.

Using the `getDefinitionObject()` method, the `createProduct()` method finds the entity definition for the `Product` entity.

2. Create a new instance.

Using the `createInstance2()` method on the entity definition, the `createProduct()` method creates a new instance of the entity object.

Note: The method name has a 2 at the end. The regular `createInstance()` method has `protected` access and is designed to be customized as described [Section E.2.4, "EntityImpl Class"](#) of [Appendix E, "Most Commonly Used ADF Business Components Methods"](#). The second argument of type `AttributeList` is used to supply attribute values that *must* be supplied at create time; it is *not* used to initialize the values of all attributes found in the list. For example, when creating a new instance of a composed child entity row using this API, you must supply the value of a composing parent entity's foreign key attribute in the `AttributeList` object passed as the second argument. Failure to do so results in an `InvalidOwnerException`.

3. Set attribute values.

Using the attribute setter methods on the entity object, the `createProduct()` method assigns values for the `Name`, `Status`, and other attributes in the new entity row.

4. Commit the transaction

Calling `commit()` on the current transaction object, the `createProduct()` method commits the transaction.

5. Return the trigger-assigned product Id to the caller

Using the attribute getter method to retrieve the value of the `ProductId` attribute as a `DBSequence`, and then calling `getSequenceNumber().longValue()`, the `createProduct()` method returns the sequence number as a `long` value to the caller.

Example 4–12 Creating a New Entity Row

```
/* Create a new Product and Return its new id */
public long createProduct(String name, String status, String shipCode) {
    //1. Find the entity definition for the Product entity
    EntityDefImpl productDef = ProductBaseEOImpl.getDefinitionObject();
    //2. Create a new instance of a Product entity
    ProductBaseEOImpl newProduct =
        (ProductBaseEOImpl)productDef.createInstance2(getDBTransaction(),null);
    //3. Set attribute values
    newProduct.setProductName(name);
    newProduct.setProductStatus(status);
    newProduct.setShippingClassCode(shipCode);
    newProduct.setSupplierId(new Number(100));
    newProduct.setListPrice(new Number(499));
    newProduct.setMinPrice(new Number(479));
    newProduct.setCreatedBy("Test Client");
    newProduct.setLastUpdatedBy("Test Client");
    newProduct.setCategoryId(new Number(5));
    //4. Commit the transaction
    try {
        getDBTransaction().commit();
    }
    catch (JboException ex) {
        getDBTransaction().rollback();
        throw ex;
    }
    //5. Access the database-trigger-assigned ProductId value and return it
    DBSequence newIdAssigned = newProduct.getProductId();
    return newIdAssigned.getSequenceNumber().longValue();
}
```

4.11.5 Assigning the Primary Key Value Using an Oracle Sequence

As an alternative to using a trigger-assigned value (as described in [Section 4.10.10, "How to Get Trigger-Assigned Primary Key Values from a Database Sequence"](#)), you can assign the value to a primary key when creating a new row using an Oracle sequence. This metadata-driven approach allows you to centralize the code to retrieve the primary key into a single Java file that can be reused by multiple entity objects.

[Example 4–13](#) shows a simple `CustomEntityImpl` framework extension class on which the entity objects are based. Its overridden `create()` method tests for the presence of a custom attribute-level metadata property named `SequenceName` and if detected, populates the attribute's default value from the next number in that sequence.

Example 4–13 CustomEntityImpl Framework Extension Class

```
package sample;

import oracle.jbo.AttributeDef;
import oracle.jbo.AttributeList;
import oracle.jbo.server.EntityImpl;
import oracle.jbo.server.SequenceImpl;
```

```

public class CustomEntityImpl extends EntityImpl {
    protected void create(AttributeList attributeList) {
        super.create(attributeList);
        for (AttributeDef def : getEntityDef().getAttributeDefs()) {
            String sequenceName = (String)def.getProperty("SequenceName");
            if (sequenceName != null) {
                SequenceImpl s = new SequenceImpl(sequenceName,getDBTransaction());
                setAttribute(def.getIndex(),s.getSequenceNumber());
            }
        }
    }
}

```

To assign the primary key value using an Oracle sequence:

1. Create the CustomEntityImpl.java file in your project, and insert the code shown in [Example 4-13](#).
2. In the Application Navigator, double-click the entity you want to edit to open the overview editor.
3. In the overview editor, click the **Attributes** tab, and double-click the attribute you want to edit.
4. In the Edit Attribute dialog, set the attribute **Type** to **Number**, and then click the **Custom Properties** node.
5. Enter SequenceName for the name.
6. Enter the name of the database sequence for the value, click **Add**, then click **OK** to create the custom property.

For example, a Dept entity could define the custom property SequenceName on its Deptno attribute with the value DEPT_TABLE_SEQ.

4.12 Generating Custom Java Classes for an Entity Object

As described in this chapter, all of the database interaction and a large amount of declarative runtime functionality of an entity object can be achieved without using custom Java code. When you need to go beyond the declarative features to implement custom business logic for your entities, you'll need to enable custom Java generation for the entities that require custom code. [Appendix E, "Most Commonly Used ADF Business Components Methods"](#), provides a quick reference to the most common code that you will typically write, use, and override in your custom entity object and entity definition classes.

4.12.1 How to Generate Custom Classes

To enable the generation of custom Java classes for an entity object, use the Java page of the overview editor.

To generate a custom Java class for an entity object:

1. In the Application Navigator, double-click the entity to open the overview editor.
2. In the overview editor, click the **Java** tab, then click the **Edit** icon.
3. In the Select Java Options dialog, select the types of Java classes you want to generate.

- Entity Object Class — the most frequently customized, it represents each row in the underlying database table.
- Entity Collection Class — rarely customized.
- Entity Definition Class — less frequently customized, it represents the related class that manages entity rows and defines their structure.

4. Click **OK**.

4.12.2 What Happens When You Generate Custom Classes

When you select one or more custom Java classes to generate, JDeveloper creates the Java file(s) you've indicated. For example, assuming an entity object named `fodemo.storefront.entities.OrderEO`, the default names for its custom Java files will be `OrderEOImpl.java` for the entity object class and `OrderEODefImpl.java` for the entity definition class. Both files are created in the same `./fodemo/storefront/entities` directory as the component's XML component definition file.

The Java generation options for the entity object continue to be reflected on subsequent visits to the Java page of the overview editor. Just as with the XML definition file, JDeveloper keeps the generated code in your custom Java classes up to date with any changes you make in the editor. If later you decide you didn't require a custom Java file for any reason, disabling the relevant options on the Java page causes the custom Java files to be removed.

4.12.3 What Happens When You Generate Entity Attribute Accessors

When you enable the generation of a custom entity object class, if you also enable the **Accessors** option, then JDeveloper generates getter and setter methods for each attribute in the entity object. For example, an `OrderEO` entity object that has the corresponding custom `OrderEOImpl.java` class might have methods (like those shown in [Example 4-14](#)) generated in it.

Example 4-14 Getter and Setter Methods from `OrderEOImpl.java`

```
public DBSequence getOrderID() { ... }
public void setOrderID(DBSequence value) { ... }

public Date getOrderDate() { ... }
public void setOrderDate(Date value) { ... }

public String getOrderStatusCode() { ... }
public void setOrderStatusCode(String value) { ... }

public Number getCustomerId() { ... }
public void setCustomerId(Number value) { ... }

public String getShipToName() { ... }
public void setShipToName(String value) { ... }
```

These methods allow you to work with the row data with compile-time checking of the correct data type usage. That is, instead of writing a line like this to get the value of the `CustomerId` attribute:

```
Number customerId = (Number)order.getAttribute("CustomerId");
```

you can write the code like:

```
Number customerId = order.getCustomerId();
```

You can see that with the latter approach, the Java compiler would catch a typographical error had you accidentally typed `CustomerCode` instead of `CustomerId`:

```
// spelling name wrong gives compile error
Number customerId = order.getCustomerCode();
```

Without the generated entity object accessor methods, an incorrect line of code like the following cannot be caught by the compiler:

```
// Both attribute name and type cast are wrong, but compiler cannot catch it
String customerId = (String)order.getAttribute("CustomerCode");
```

It contains both an incorrectly spelled attribute name, as well as an incorrectly typed cast of the `getAttribute()` return value. When you use the generic APIs on the `Row` interface, which the base `EntityImpl` class implements, errors of this kind raise exceptions at runtime instead of being caught at compile time.

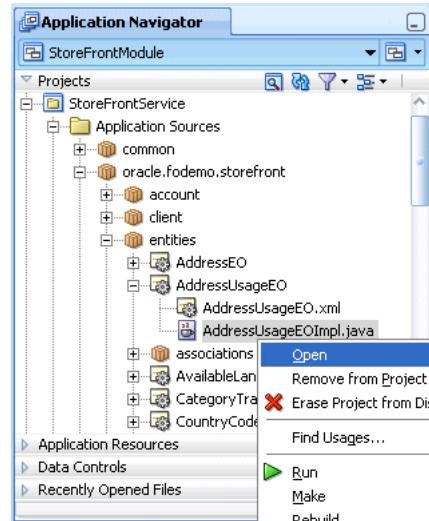
4.12.4 How to Navigate to Custom Java Files

As shown in [Figure 4–13](#), when you've enabled generation of custom Java classes, they also appear under the **Application Sources** node for the entity object, as child nodes.

As with all ADF components, when you select an entity object in the Application Navigator, the Structure window provides a structural view of the entity. When you need to see or work with the source code for a custom Java file, there are two ways to open the file in the source editor:

- You can right-click the Java file, and choose **Open** from the context menu, as shown in [Figure 4–13](#).
- You can right-click an item in a node in the Structure window, and choose **Go To Source** from the context menu.

Figure 4–13 Seeing and Navigating to Custom Java Classes for an Entity Object



4.12.5 What You May Need to Know About Custom Java Classes

See the following sections for additional information about custom Java classes.

4.12.5.1 About the Framework Base Classes for an Entity Object

When you use an XML-only entity object, at runtime its functionality is provided by the default ADF Business Components implementation classes. Each custom Java class that is generated will automatically extend the appropriate ADF Business Components base class so that your code inherits the default behavior and you can easily add to or customize it. An entity object class will extend `EntityImpl`, while the entity definition class will extend `EntityDefImpl` (both in the `oracle.jbo.server` package).

4.12.5.2 You Can Safely Add Code to the Custom Component File

Some developers are hesitant to add their own code to generated Java source files. Each custom Java source code file that JDeveloper creates and maintains for you includes the following comment at the top of the file to clarify that it is safe for you to add your own custom code to this file.

```
// -----
// ---  File generated by Oracle ADF Business Components Design Time.
// ---  Custom code may be added to this class.
// ---  Warning: Do not modify method signatures of generated methods.
// -----
```

JDeveloper does not blindly regenerate the file when you click the **OK** or **Apply** button in the component editor. Instead, it performs a smart update to the methods that it needs to maintain, leaving your own custom code intact.

4.12.5.3 Configuring Default Java Generation Preferences

You can generate custom Java classes for your view objects when you need to customize their runtime behavior or when you simply prefer to have strongly typed access to bind variables or view row attributes.

To configure the default settings for ADF Business Components custom Java generation, you can choose **Preferences** from the **Tools** menu and open the Business Components page to set your preferences to be used for business components created in the future. Developers getting started with ADF Business Components should set their preference to generate no custom Java classes by default. As you run into a specific need for custom Java code, you can enable just the bit of custom Java you need for that one component. Over time, you'll discover which set of defaults works best for you.

4.12.5.4 Attribute Indexes and InvokeAccessor Generated Code

The entity object is designed to function based on XML only or as an XML component definition combined with a custom Java class. To support this design choice, attribute values are not stored in private member fields of an entity's class (a file that is not present in the XML-only situation). Instead, in addition to a name, attributes are also assigned a numerical index in the entity's XML component definition based on the zero-based, sequential order of the `<Attribute>` and association-related `<AccessorAttribute>` tags in that file. At runtime, attribute values in an entity row are stored in a sparse array structure managed by the base `EntityImpl` class, indexed by the attribute's numerical position in the entity's attribute list.

For the most part, this private implementation detail is unimportant, since as a developer using entity objects, you are shielded from having to understand this. However, when you enable a custom Java class for your entity object, this implementation detail relates to some of the generated code that JDeveloper maintains in your entity object class. It is sensible to understand what that code is used for. For

example, in the custom Java class for a OrderEO entity object, each attribute or accessor attribute has a corresponding generated integer constant, as shown in [Example 4-15](#). JDeveloper ensures that the values of these constants correctly reflect the ordering of the attributes in the XML component definition.

Example 4-15 Attribute Constants in the Custom Entity Java Class

```
public class OrderEOImpl extends EntityImpl {
    public static final int ORDERID = 0;
    public static final int ORDERDATE = 1;
    public static final int ORDERSHIPPEDDATE = 2;
    public static final int ORDERSTATUSCODE = 3;
    public static final int ORDERTOTAL = 4;
    public static final int CUSTOMERID = 5;
    public static final int SHIPTONAME = 6;
    public static final int SHIPTOADDRESSID = 7;
    public static final int SHIPTOPHONENUMBER = 8;
    public static final int SHIPPINGOPTIONID = 9;
    public static final int PAYMENTOPTIONID = 10;
    // etc.
```

You'll also notice that the automatically maintained, strongly typed getter and setter methods in the entity object class use these attribute constants, as shown in [Example 4-16](#).

Example 4-16 Getter and Setter Methods Using Attribute Constants in the Custom Entity Java Class

```
// In oracle.fodemo.storefront.entities.OrderEOImpl class
public Date getOrderDate() {
    return (Date)getAttributeInternal(ORDERDATE); // <-- Attribute constant
}
public void setOrderDate(Date value) {
    setAttributeInternal(ORDERDATE, value); // <-- Attribute constant
}
```

Another aspect of the automatically maintained code related to entity attribute constants are the `getAttrInvokeAccessor()` and `setAttrInvokeAccessor()` methods. These methods optimize the performance of attribute access by numerical index, which is how generic code in the `EntityImpl` base class typically accesses attribute values when performing generic processing. An example of the `getAttrInvokeAccessor()` method is shown in [Example 4-17](#). The companion `setAttrInvokeAccessor()` method looks similar.

Example 4-17 getAttrInvokeAccessor() Method in the Custom Entity Java Class

```
// In oracle.fodemo.storefront.entities.OrderEOImpl class
/** getAttrInvokeAccessor: generated method. Do not modify. */
protected Object getAttrInvokeAccessor(int index, AttributeDefImpl attrDef)
throws Exception {
    switch (index) {
    case ORDERID:
        return getOrderId();
    case ORDERDATE:
        return getOrderDate();
    case ORDERSHIPPEDDATE:
        return getOrderShippedDate();
    case ORDERSTATUSCODE:
        return getOrderStatusCode();
    case ORDERTOTAL:
```

```
        return getOrderTotal();
    case CUSTOMERID:
        return getCustomerId();
    case SHIPTONAME:
        return getShipToName();
    ...
    default:
        return super.getAttrInvokeAccessor(index, attrDef);
    }
}
```

The rules of thumb to remember about this generated attribute-index related code are the following.

The Do's

- Add custom code if needed inside the strongly typed attribute getter and setter methods.
- Use the overview editor to change the order or type of entity object attributes.
JDeveloper changes the Java signature of getter and setter methods, as well as the related XML component definition for you.

The Don'ts

- Don't modify the `getAttrInvokeAccessor()` and `setAttrInvokeAccessor()` methods.
- Don't change the values of the attribute index numbers manually.

Note: If you need to manually edit the generated attribute constants because of source control merge conflicts or other reasons, you must ensure that the zero-based ordering reflects the sequential ordering of the `<Attribute>` and `<AccessorAttribute>` tags in the corresponding entity object XML component definition.

4.12.6 Programmatic Example for Comparison Using Custom Entity Classes

To better evaluate the difference of using custom generated entity classes versus working with the generic `EntityImpl` class, [Example 4-18](#) shows a version of methods in a custom entity class (`StoreFrontServiceImpl.java`) from a custom application module class (`StoreFrontService2Impl.java`). Some important differences to notice are:

- Attribute access is performed using strongly typed attribute accessors.
- Association accessor attributes return the strongly typed entity class on the other side of the association.
- Using the `getDefinitionObject()` method in your custom entity class allows you to avoid working with fully qualified entity definition names as strings.
- The `createPrimaryKey()` method in your custom entity class simplifies creating the `Key` object for an entity.

Example 4-18 Programmatic Entity Examples Using Strongly Typed Custom Entity Object Classes

```
package devguide.examples.appmodules;

import oracle.fodemo.storefront.entities.OrderEOImpl;
```

```

import oracle.fodemo.storefront.entities.PersonEOImpl;
import oracle.fodemo.storefront.entities.ProductBaseEOImpl;

import oracle.jbo.ApplicationModule;
import oracle.jbo.JboException;
import oracle.jbo.Key;
import oracle.jbo.client.Configuration;
import oracle.jbo.domain.DBSequence;
import oracle.jbo.domain.Number;
import oracle.jbo.server.ApplicationModuleImpl;
import oracle.jbo.server.EntityDefImpl;

// -----
// --- File generated by Oracle ADF Business Components Design Time.
// --- Custom code may be added to this class.
// --- Warning: Do not modify method signatures of generated methods.
// -----
/***
 * This custom application module class illustrates the same
 * example methods as StoreFrontServiceImpl.java, except that here
 * we're using the strongly typed custom Entity Java classes
 * OrderEOImpl, PersonsEOImpl, and ProductsBaseEOImpl instead of working
 * with all the entity objects using the base EntityImpl class.
 */

public class StoreFrontService2Impl extends ApplicationModuleImpl {
    /**This is the default constructor (do not remove).
     */
    public StoreFrontService2Impl() {
    }
    /**
     * Helper method to return an Order by Id
     */
    private OrderEOImpl retrieveOrderById(long orderId) {
        EntityDefImpl orderDef = OrderEOImpl.getDefinitionObject();
        Key orderKey = OrderEOImpl.createPrimaryKey(new DBSequence(orderId));
        return (OrderEOImpl)orderDef.findByPrimaryKey(getDBTransaction(),orderKey);
    }

    /**
     * Find an Order by Id
     */
    public String findOrderTotal(long orderId) {
        OrderEOImpl order = retrieveOrderById(orderId);
        if (order != null) {
            return order.getOrderTotal().toString();
        }
        return null;
    }

    /**
     * Create a new Product and Return its new id
     */
    public long createProduct(String name, String status, String shipCode) {
        EntityDefImpl productDef = ProductBaseEOImpl.getDefinitionObject();
        ProductBaseEOImpl newProduct =
(ProductBaseEOImpl)productDef.createInstance2(getDBTransaction(),null);
        newProduct.setProductName(name);
        newProduct.setProductStatus(status);
    }
}

```

```
        newProduct.setShippingClassCode(shipCode);
        newProduct.setSupplierId(new Number(100));
        newProduct.setListPrice(new Number(499));
        newProduct.setMinPrice(new Number(479));
        newProduct.setCreatedBy("Test Client");
        newProduct.setLastUpdatedBy("Test Client");
        newProduct.setCategoryId(new Number(5));
    try {
        getDBTransaction().commit();
    }
    catch (JboException ex) {
        getDBTransaction().rollback();
        throw ex;
    }
    DBSequence newIdAssigned = newProduct.getProductId();
    return newIdAssigned.getSequenceNumber().longValue();
}
/*
 * Update the status of an existing order
 */
public void updateRequestStatus(long orderId, String newStatus) {
    OrderEOImpl order = retrieveOrderById(orderId);
    if (order != null) {
        order.setOrderStatusCode(newStatus);
        try {
            getDBTransaction().commit();
        }
        catch (JboException ex) {
            getDBTransaction().rollback();
            throw ex;
        }
    }
}

/*
 * Access an associated Customer entity from the Order entity
 */
public String findOrderCustomer(long orderId) {
    OrderEOImpl svcReq = retrieveOrderById(orderId);
    if (svcReq != null) {
        PersonEOImpl cust = (PersonEOImpl) svcReq.getPerson();
        if (cust != null) {
            return cust.getFirstName() + " " + cust.getLastName();
        }
        else {
            return "Unassigned";
        }
    }
    else {
        return null;
    }
}

/*
 * Testing method
 */
public static void main(String[] args) {
    String amDef = "devguide.model.StoreFrontService";
    String config = "StoreFrontServiceLocal";
    ApplicationModule am =
```

```

Configuration.createRootApplicationModule(amDef, config);
/*
 * NOTE: This cast to use the StoreFrontServiceImpl class is OK since
 * this code is inside a business tier *Impl.java file and not in a
 * client class that is accessing the business tier from "outside".
 */
StoreFrontServiceImpl service = (StoreFrontServiceImpl)am;
String total = service.findOrderTotal(1011);
System.out.println("Status of Order # 1011 = " + total);
String customerName = service.findOrderCustomer(1011);
System.out.println("Customer for Order # 1011 = " + customerName);
try {
    service.updateOrderStatus(1011, "CANCEL");
}
catch (JboException ex) {
    System.out.println("ERROR: "+ex.getMessage());
}
long id = 0;
try {
    id = service.createProduct(null, "NEW", "CLASS1");
}
catch (JboException ex) {
    System.out.println("ERROR: "+ex.getMessage());
}
id = service.createProduct("Canon PowerShot G9", "NEW", "CLASS1");
System.out.println("New product created successfully with id = "+id);
Configuration.releaseRootApplicationModule(am,true);
}
}
}

```

4.13 Adding Transient and Calculated Attributes to an Entity Object

In addition to having attributes that map to columns in an underlying table, your entity objects can include transient attributes that display values calculated (for example, using Java or Groovy) or that are value holders. For example, a transient attribute you create, such as `FullName`, could be calculated based on the concatenated values of `FirstName` and `LastName` attributes.

Once you create the transient attribute, you can perform a calculation in the entity object Java class, or use a Groovy expression in the attribute definition to specify a default value.

If you want to be able to change the value at runtime, you can use a Groovy expression. If the calculated value is not likely to change (for example, if it's a sum of the line items), you can perform the calculation directly in the entity object Java class.

4.13.1 How to Add a Transient Attribute

Use the Attributes page of the overview editor to create a transient attribute.

To add a transient attribute to an entity object:

1. Open the Attributes page in the overview editor and click the **New** icon.
2. Enter a name for the attribute.
3. Set the Java attribute type.
4. Disable the **Persistent** option.
5. If the value will be calculated, set **Updatable** to **Never**.

6. Click **OK**.

4.13.2 What Happens When You Add a Transient Attribute

When you add a transient attribute, JDeveloper updates the XML component definition for the entity object to reflect the new attribute.

The `<Attribute>` tag of a transient attribute has no `TableName` and a `ColumnName` of `$none$`, as shown in [Example 4–19](#).

Example 4–19 XML Code for a Transient Attribute

```
<Attribute
    Name="FullName"
    IsUpdateable="false"
    IsQueryable="false"
    IsPersistent="false"
    ColumnName="$none$"
    Type="java.lang.String"
    ColumnType="$none$"
    SQLType="VARCHAR" >
</Attribute>
```

In contrast, a persistent entity attribute has both a `TableName` and a `ColumnName`, as shown in [Example 4–20](#).

Example 4–20 XML Code for a Persistent

```
<Attribute
    Name="FirstName"
    IsNotNull="true"
    Precision="30"
    ColumnName="FIRST_NAME"
    Type="java.lang.String"
    ColumnType="VARCHAR2"
    SQLType="VARCHAR"
    TableName="USERS" >
</Attribute>
```

4.13.3 How to Base a Transient Attribute On a Groovy Expression

When creating a transient attribute, you can use a Groovy expression to provide the default value.

To create a transient attribute based on a Groovy expression:

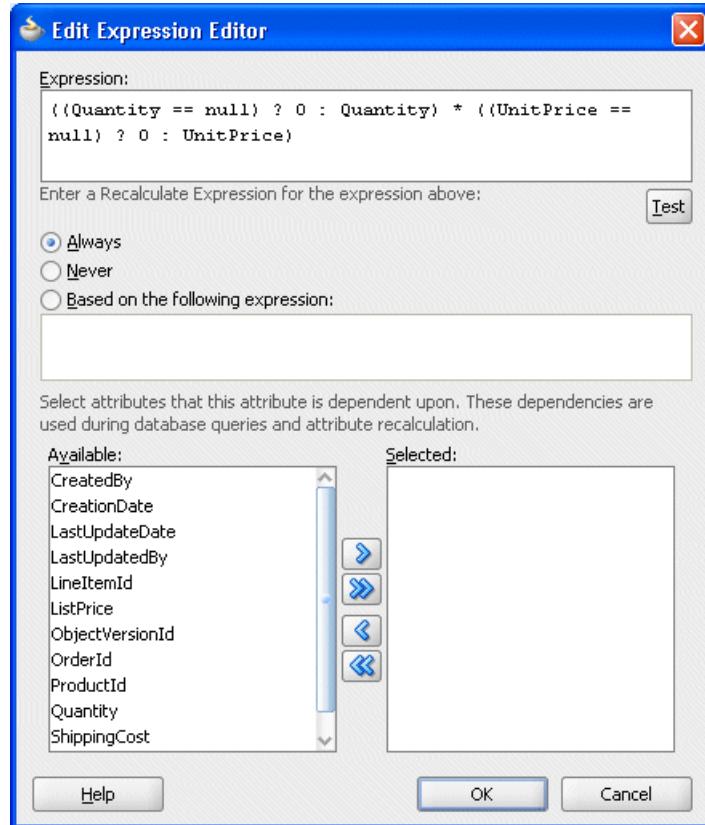
1. Create a new attribute, as described in the first four steps of [Section 4.13.1, "How to Add a Transient Attribute"](#).
2. In the New Entity Attribute dialog box, click the **Edit** button next to the **Value** field.

Expressions that you define are evaluated using the Groovy Expression Language, as described in [Section 3.6, "Overview of Groovy Support"](#). Groovy lets you insert expressions and variables into strings. The expression is saved as part of the entity object definition

3. In the Edit Expression dialog, enter an expression in the field provided, as shown in [Figure 4–14](#).

Attributes that you reference can include any attribute that the entity object define. Do not reference attributes in the expression that are not defined by the entity object.

Figure 4–14 Edit Expression Dialog



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. You can optionally provide a condition for when to recalculate the expression.

For example, the following expression in the **Based on the following expression** field causes the attribute to be recalculated when either the **Quantity** attribute or the **UnitPrice** attribute are changed:

```
return (adf.object.isAttributeChanged("Quantity") ||
    adf.object.isAttributeChanged("UnitPrice"));
```

6. Click **OK** to save the expression.

7. Then click **OK** to create the attribute.

Note: If either the value expression or the optional recalculate expression that you define references an attribute from the base entity object, you must define this as a dependency on the Dependencies page of the Edit Attribute dialog. In the Dependency page, locate the attributes in the **Available** list and shuttle each to the **Selected** list.

4.13.4 What Happens When You Base a Transient Attribute on Groovy Expression

When you base a transient attribute on a Groovy expression, a `<TransientExpression>` tag is added to the entity object's XML file within the appropriate attribute, as shown in [Example 4–21](#).

Example 4–21 Calculating a transient attribute using a Groovy expression

```
<TransientExpression>
  <![CDATA[
    ((Quantity == null) ? 0 : Quantity) * ((UnitPrice == null) ? 0 : UnitPrice)
  ]]>
</TransientExpression>
```

4.13.5 How to Add Java Code in the Entity Class to Perform Calculation

A transient attribute is a placeholder for a data value. If you change the **Updatable** property of the transient attribute to **While New** or **Always**, then the end user can enter a value for the attribute. If you want the transient attribute to display a calculated value, then you'll typically leave the **Updatable** property set to **Never** and write custom Java code that calculates the value.

After adding a transient attribute to the entity object, to make it a *calculated* attribute you need to:

- Enable a custom entity object class on the Java page of the overview editor, choosing to generate accessor methods
- Write Java code inside the accessor method for the transient attribute to return the calculated value
- Specify each dependent attribute for the transient attribute on the Dependencies page of the Edit Attribute dialog

For example, after generating the view row class, the Java code to return the transient attribute's calculated value would reside in the getter method for the attribute (such as `FullName`), as shown in [Example 4–22](#).

Example 4–22 Getter Method for a Transient Attribute

```
// Getter method for FullName calculated attribute in UserImpl.java
public String getFullName() {
    // Commented out original line since we'll always calculate the value
    // return (String)getAttributeInternal(FULLNAME);
    return getFirstName()+" "+getLastName();
}
```

To ensure that the transient attribute is reevaluated whenever the attributes to be concatenated (such as `LastName` and `FirstName`) might be changed by the end user, specify the dependent attributes for the transient attribute. On the Dependencies page of the Edit Attribute dialog, locate the attributes in the **Available** list and shuttle each to the **Selected** list.

Defining SQL Queries Using View Objects

This chapter describes how to use the Create View Object wizard and the various view object editors to join, filter, sort, and aggregate data for use in the application.

This chapter includes the following sections:

- [Section 5.1, "Introduction to View Objects"](#)
- [Section 5.2, "Populating View Object Rows from a Single Database Table"](#)
- [Section 5.3, "Populating View Object Rows with Static Data"](#)
- [Section 5.4, "Limiting View Object Rows Using Effective Date Ranges"](#)
- [Section 5.5, "Working with Multiple Tables in Join Query Results"](#)
- [Section 5.6, "Working with Multiple Tables in a Master-Detail Hierarchy"](#)
- [Section 5.7, "Working with View Objects in Declarative SQL Mode"](#)
- [Section 5.8, "Working with View Objects in Expert Mode"](#)
- [Section 5.9, "Working with Bind Variables"](#)
- [Section 5.10, "Working with Named View Criteria"](#)
- [Section 5.11, "Working with List of Values \(LOV\) in View Object Attributes"](#)
- [Section 5.12, "Defining Attribute Control Hints for View Objects"](#)
- [Section 5.13, "Adding Calculated and Transient Attributes to a View Object"](#)

5.1 Introduction to View Objects

A *view object* is an Oracle ADF component that encapsulates a SQL query and simplifies working with its results. There are several types of view objects that you can create in your ADF Business Components project:

- Read-only view objects when updates to data are not necessary (may be entity-based)
- Entity-based view objects when data updates will be performed
- Static data view objects for data defined by the view object itself
- Programmatically populated view objects (for more information, see [Chapter 35, "Advanced View Object Techniques"](#))

An *entity-based view object* can be configured to support updatable rows when you create view objects that map their attributes to the attributes of one or more existing entity objects. The mapped entity object is saved as an entity usage in the view object definition. In this way, entity-based view object cooperate automatically with entity

objects to enable a fully updatable data model. The entity-based view object queries just the data needed for the client-facing task and relies on its mapped entity objects to automatically validate and save changes made to its view rows. Like the read-only view object, an entity-based view object encapsulates a SQL query, it can be linked into master-detail hierarchies, and it can be used in the data model of your application modules.

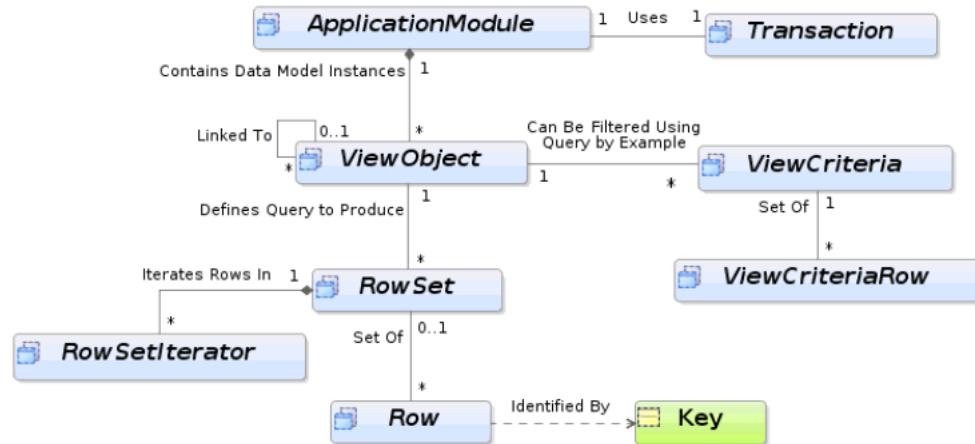
5.1.1 Overview of View Object Concepts

View objects with no entity usage definition are always read-only. They do not pick up entity-derived default values, they do not reflect pending changes, and they do not reflect updated reference information. In contrast to entity-based view objects, *read-only view objects* require you to write the query using the SQL query language. The Create View Object wizard and overview editor for entity-based view objects, on the other hand, simplify this task by helping you to construct the SQL query declaratively. For this reason, it is almost always preferable to create a non-updatable, entity-mapped view object, even when you want to create a view object just to read data. Additionally, as an alternative to creating view objects that specify a SQL statement at design time, you can create entity-mapped view objects that dynamically generate SQL statements at runtime.

There remain a few situations where it is still preferable to create a non-entity-mapped view object to read data, including SQL-based validation, Unions, and Group By queries.

This chapter helps you understand these view object concepts as illustrated in [Figure 5-1](#):

- You define a view object by providing a SQL query (either defined explicitly or declaratively).
- You use view object instances in the context of an application module that provides the database transaction for their queries.
- You can link a view object to one or more others to create master-detail hierarchies.
- At runtime, the view object executes your query and produces a set of rows (represented by a `RowSet` object).
- Each row is identified by a corresponding row key.
- You iterate through the rows in a row set using a row set iterator.
- You can filter the row set a view object produces by applying a set of query-by-example criteria rows.

Figure 5–1 A View Object Defines a Query and Produces a Row Set of Rows

5.1.2 Runtime Features Unique to Entity-Based View Objects

When a view object has one or more underlying entity usages, you can create new rows, and modify or remove queried rows. The entity-based view object coordinates with underlying entity objects to enforce business rules and to permanently save the changes to the database. In addition, entity-based view objects provide these capabilities that do not exist with read-only view objects:

- Changes in cache (updates, inserts, deletes) managed by entities survive the view object's execution boundary
- Changes made to relevant entity object attributes through other view objects in the same transaction are immediately reflected
- Attribute values of new rows are initialized to the values from the underlying entity object attributes
- Changes to foreign key attribute values cause reference information to get updated
- Validation for row (entity) level is supported
- Composition feature, including validation, locking, ordered-updates is supported
- Effective dating and change indicator is supported

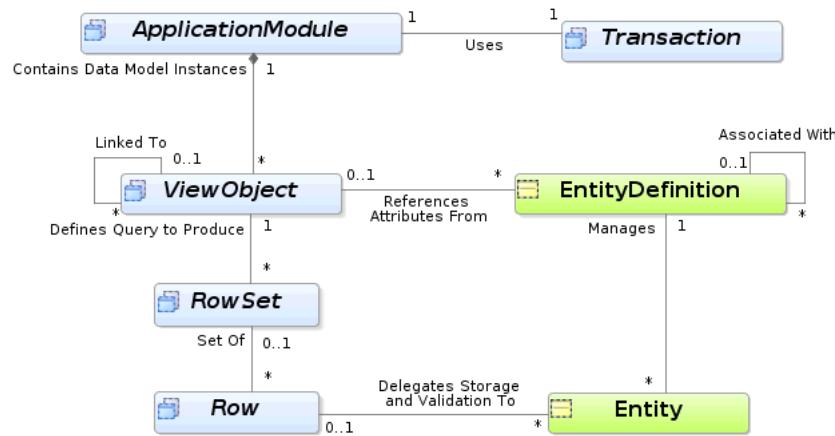
This chapter explains how instances of entity-based view objects contained in the data model of your application module enable clients to search for, update, insert, and delete business domain layer information in a way that combines the full data shaping power of SQL with the clean, object-oriented encapsulation of reusable domain business objects. And all without requiring a line of code.

This chapter helps you to understand these entity-based view object concepts as illustrated in [Figure 5–2](#):

- You define an updatable view object by referencing attributes from one or more entity objects.
- You can use multiple, associated entity objects to simplify working with reference information.
- You can define view links based on underlying entity associations.
- You use your entity-based view objects in the context of an application module that provides the transaction.

- At runtime, the view row delegates the storage and validation of its attributes to underlying entity objects.

Figure 5–2 View Objects and Entity Objects Collaborate to Enable an Updatable Data Model



5.2 Populating View Object Rows from a Single Database Table

View objects provide the means to retrieve data from a data source. In the majority of cases, the data source will be a database and the mechanism to retrieve data is the SQL query. ADF Business Components can work with JDBC to pass this query to the database and retrieve the result.

When view objects use a SQL query, query columns map to view object attributes in the view object. The definition of these attributes, saved in the view object's XML definition file, reflect the properties of these columns, including data types and precision and scale specifications.

Performance Tip: If the query associated with the view object contains values that may change from execution to execution, use bind variables. Using bind variables in the query allows the query to reexecute without needing to reparse the query on the database. You can add bind variables to the view object in the Query page of the overview editor for the view object. For more information, see [Section 5.9, "Working with Bind Variables"](#).

Using the same Create View Object wizard, you can create view objects that either map to the attributes of existing entity objects or not. Only entity-based view objects automatically coordinate with mapped entity objects to enforce business rules and to permanently save data model changes. Additionally, you can disable the Updatable feature for entity-based view objects and work entirely declaratively to query read-only data. Alternatively, you can use the wizard or editor's expert mode to work directly with the SQL query language, but the view object you create will not support the transaction features of the entity-based view object.

While there is a small amount of runtime overhead associated with the coordination between view object rows and entity object rows, weigh this against the ability to keep the view object definition entirely declarative and maintain a customizable view object. Queries that cannot be expressed in entity objects, and that therefore require expert-mode query editing, include Unions and Group By queries. Expert mode-based view objects are also useful in SQL-based validation queries used by the view.

object-based Key Exists validator. Again, it is worth repeating that, by definition, using expert mode to define a SQL query means the view object must be read-only.

For more information about the differences between entity-based view objects and read-only view objects, see [Section 5.1.2, "Runtime Features Unique to Entity-Based View Objects"](#).

5.2.1 How to Create an Entity-Based View Object

Creating an entity-based view object is the simplest way to create a view object. It is even easier than creating an expert-mode, read-only view object, since you don't have to type in the SQL statement yourself. An entity-based view object also offers significantly more runtime functionality than its expert-mode counterpart.

In an entity-based view object, the view object and entity object play cleanly separated roles:

- The view object is the *data source*: it retrieves the data using SQL.
- The entity object is the *data sink*: it handles validating and saving data changes.

Because view objects and entity objects have cleanly separated roles, you can build a hundred different view objects — projecting, filtering, joining, sorting the data in whatever way your user interfaces require, application after application — without any changes to the reusable entity object. In fact, it is possible that the development team responsible for the core business domain layer of entity objects might be completely separate from another team responsible for the specific application modules and view objects needed to support the end-user environment. This relationship is enabled by metadata that the entity-based view object encapsulates. The metadata specifies how the SELECT list columns related to the attributes of one or more underlying entity objects.

5.2.1.1 Creating an Entity-Based View Object from a Single Table

To create an entity-based view object, use the Create View Object wizard, which is available from the New Gallery.

Your entity-based view object may be based on more than one database table. To use database joins to add multiple tables to the view object, see [Section 5.5, "Working with Multiple Tables in Join Query Results"](#).

To create entity-based view object from a single table:

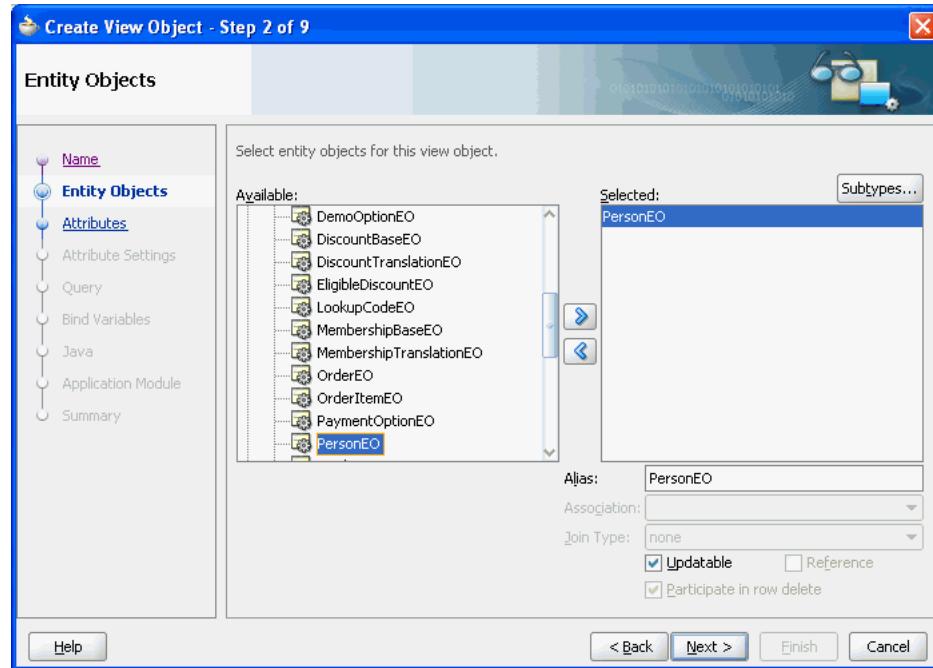
1. In the Application Navigator, right-click the project in which you want to create the view objects and choose **New**.
2. In the New Gallery, expand **Business Tier**, select **ADF Business Components**, then **View Object**, and click **OK**.

If this is the first component you're creating in the project, the Initialize Business Components Project dialog appears to allow you to select a database connection.
3. In the Initialize Business Components Project dialog, select the database connection or choose **New** to create a connection. Click **OK**.
4. In the Create View Object wizard, on the Name page, enter a package name and a view object name. Keep the default setting **Updatable Access Through Entity Objects** enabled to indicate that you want this view object to manage data with its base entity object. Click **Next**.
5. On the Entity Objects page, select an entity object whose data you want to use in the view object. Click **Next**.

An entry in this list is known as an *entity usage*, since it records the entity objects that the view object will be using. Each entry could also be thought of as an *entity reference*, since the view object references attributes from that entity. For information about working table joins to create additional entity usages, see [Section 5.5, "Working with Multiple Tables in Join Query Results"](#).

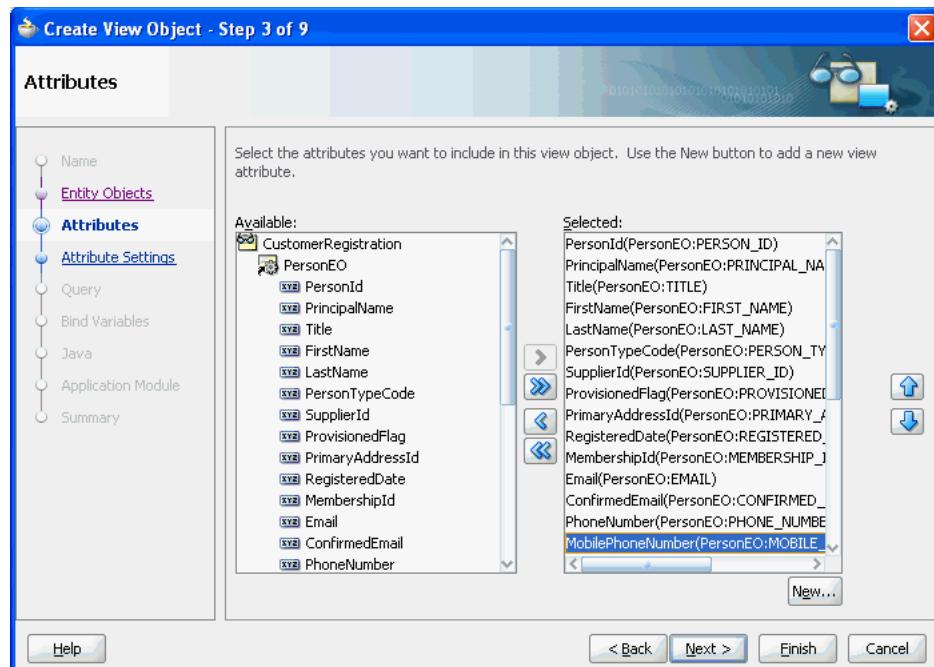
For example, [Figure 5–3](#) shows the result after shuttling the PersonEO entity object into the **Selected** list.

Figure 5–3 Create View Object Wizard, Entity Objects Page



6. On the Attributes page, select the attributes you want to include from each entity usage in the **Available** list and shuttle them to the **Selected** list. Click **Next**.

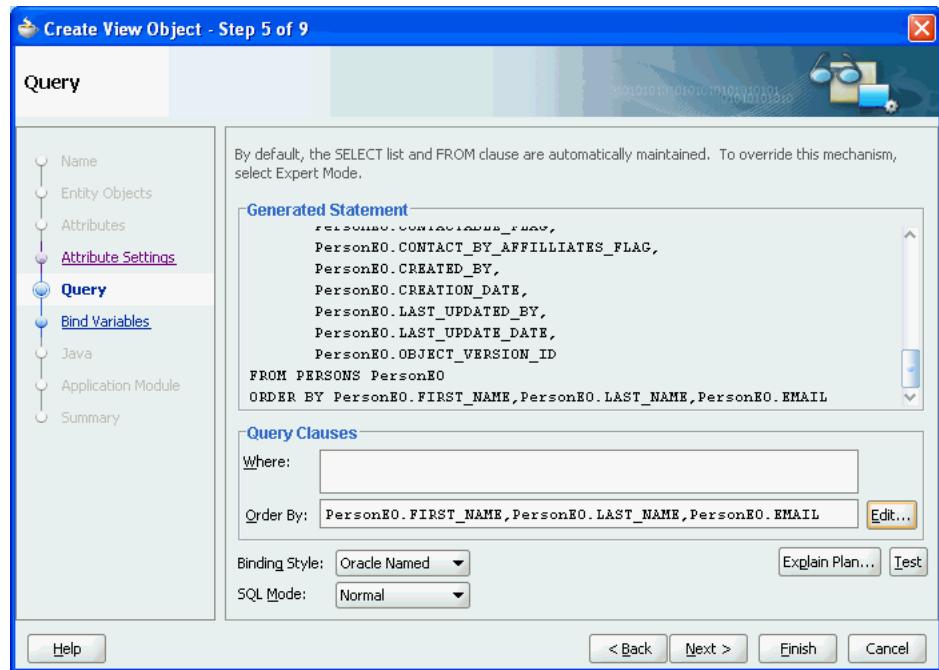
For example, [Figure 5–4](#) shows the attributes have been selected from the PersonEO.

Figure 5–4 Create View Object Wizard, Attributes Page

7. On the Attribute Settings page, optionally, use the **Select Attribute** dropdown list to switch between the view object attributes in order to change their names or any of their initial settings.
8. On the Query page, optionally, add a WHERE and ORDER BY clause to the query to filter and order the data as required. JDeveloper automatically generates the SELECT statement based on the entity attributes you've selected.

Do not include the WHERE or ORDER BY keywords in the **Where** and **Order By** field values. The view object adds those keywords at runtime when it executes the query.

For example, [Figure 5–5](#) shows the ORDER BY clause is specified to order the data by first name, last name, and email.

Figure 5–5 Create View Object Wizard, Query Page

- When you are satisfied with the view object, click **Finish**.

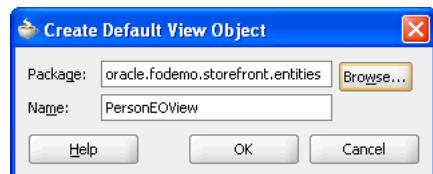
5.2.1.2 Creating a View Object with All the Attributes of an Entity Object

When you want to allow the client to work with *all* of the attributes of an underlying entity object, you can use the Create View Object wizard as described in [Section 5.2.1.1, "Creating an Entity-Based View Object from a Single Table"](#). After selecting the entity object, simply select *all* of its attributes on the Attributes page. However, for this frequent operation, there is an even quicker way to perform the same task in the Application Navigator.

To create a default entity-based view object:

- In the Application Navigator, right-click the entity object and choose **New Default View Object**.
- Provide a package and component name for the new view object in the Create Default View Object dialog.

In the Create Default View Object dialog you can click **Browse** to select the package name from the list of existing packages. For example, in [Figure 5–6](#), clicking Browse locates oracle.fodemo.storefront.entities package on the classpath for the StoreFrontService project in the StoreFrontModule application.

Figure 5–6 Shortcut to Creating a Default View Object for an Entity Object

The new entity-based view object created will be identical to one you could have created with the Create View Object wizard. By default, it will have a single entity usage referencing the entity object you selected in the Application Navigator, and will include all of its attributes. It will initially have neither a WHERE nor ORDER BY clause, and you may want to use the overview editor for the view object to:

- Remove unneeded attributes
- Refine its selection with a WHERE clause
- Order its results with an ORDER BY clause
- Customize any of the view object properties

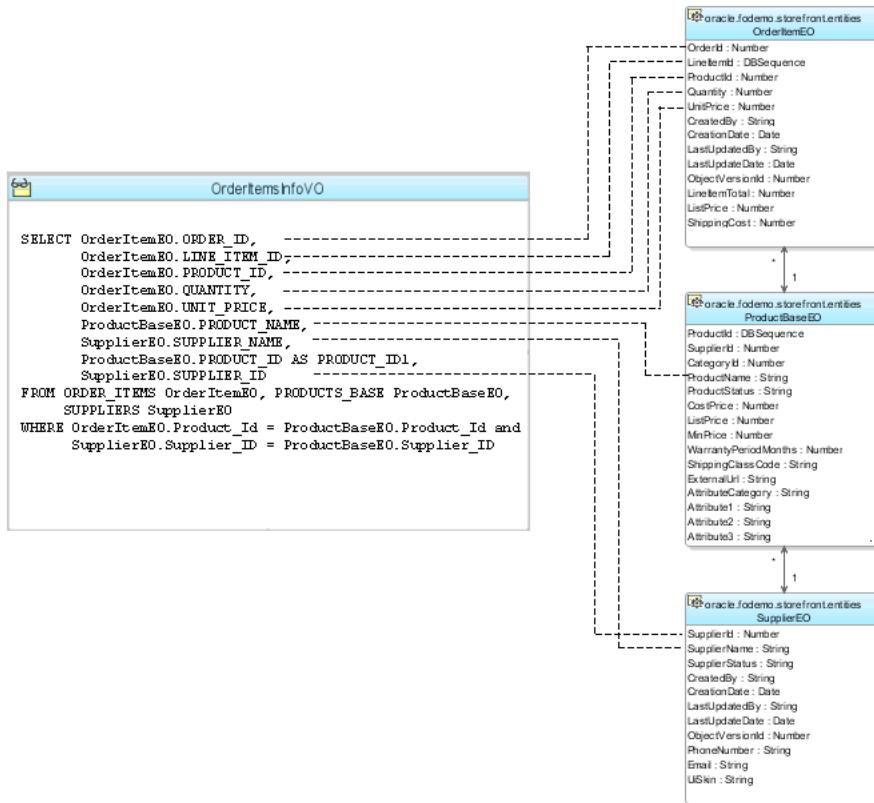
5.2.2 What Happens When You Create an Entity-Based View Object

When you create a view object, JDeveloper creates the XML component definition file that represents the view object's declarative settings and saves it in the directory that corresponds to the name of its package. For example, the view object Orders, added to the queries package, will have the XML file ./queries/Orders.xml created in the project's source path.

To view the view object settings, expand the desired view object in the Application Navigator, select the XML file under the expanded view object, and open the Structure window. The Structure window displays the list of definitions, including the SQL query, the name of the entity usage, and the properties of each attribute. To open the file in the editor, double-click the corresponding .xml node.

Note: If your IDE-level Business Components Java generation preferences so indicate, the wizard may also create an optional custom view object class OrdersImpl.java and/or a custom view row class OrdersRowImpl.java class.

The dotted lines in [Figure 5-7](#) represent the metadata captured in the entity-based view object's XML component definition that maps SELECT list columns in the query to attributes of the entity objects used in the view object. It joins data from a primary entity usage (OrderItemEO) with that from secondary reference entity usages (ProductBaseEO and SupplierEO).

Figure 5–7 View Object Encapsulates a SQL Query and Entity Attribute Mapping Metadata

5.2.3 How to Create an Expert Mode, Read-Only View Object

When you need full control over the SQL statement, the Create View Object wizard lets you specify that you want a view object to be read-only. In this case, you will not benefit from the declarative capabilities to define a non-updatable entity-based view object. However, there are a few situations where it is desirable to create read-only view objects using expert mode. Primarily, the read-only view object that you create will be useful when you need to write Unions or Group By queries. Additionally, you can use a read-only view object if you need to create SQL-based validation queries used by the view object-based Key Exists validator, provided that you have marked a key attribute.

For more information about the tradeoffs between working with entity-based view objects that you define as non-updatable and strictly read-only view objects, see [Section 35.2.2, "Consider Using Entity-Based View Objects for Read-Only Data"](#).

To create a read-only view object, use the Create View Object wizard, which is available from the New Gallery.

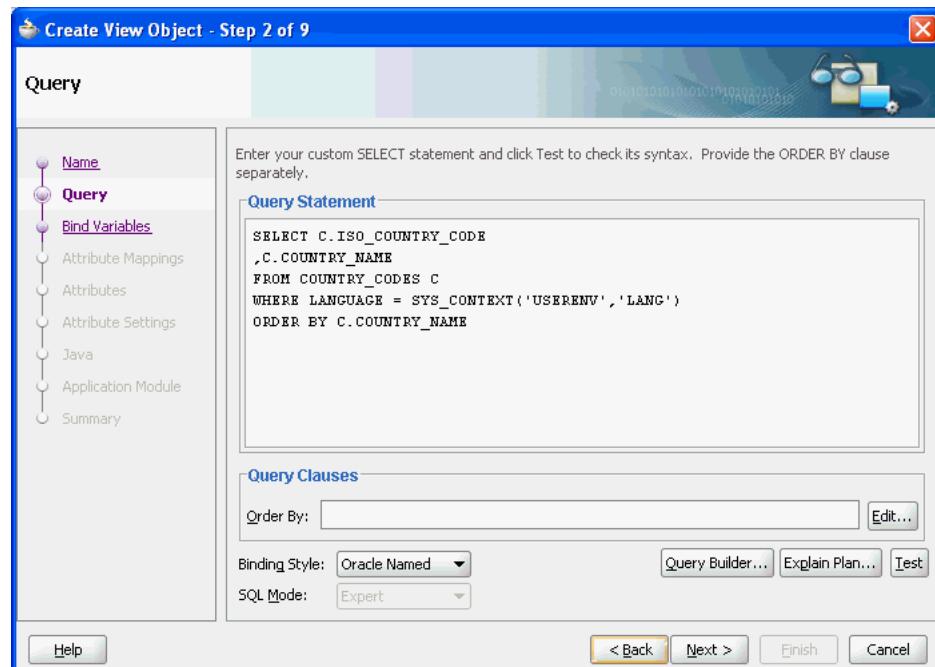
To create a read-only view object:

1. In the Application Navigator, right-click the project in which you want to create the view object and choose **New**.
2. In the New Gallery, expand **Business Tier**, select **ADF Business Components**, and then **View Object**, and click **OK**.

If this is the first component you're creating in the project, the Initialize Business Components Project dialog appears to allow you to select a database connection.

3. In the Initialize Business Components Project dialog, select the database connection or choose **New** to create a connection. Click **OK**.
4. In the Create View Object wizard, on the Name page, enter a package name and a view object name. Select **Read-only Access through SQL Query** to indicate that you want this view object to manage data with read-only access. Click **Next**.
5. On the Query page, use one of the following techniques:
 - Paste any valid SQL statement into the **Query Statement** box. The query statement can use a WHERE clause and an Order By clause. For example, [Figure 5–8](#) shows a query statement that uses a WHERE clause and an Order By clause to query a list of country codes in the language used by the application.
 - Click **Query Builder** to open the SQL Statement dialog and use the interactive query builder.

Figure 5–8 Create View Object Wizard, Query Page



Note: If the Entity Objects page displays instead of the Query page, go back to Step 1 of the wizard and ensure that you've selected **Read-only Access**.

6. After entering or building the query statement, click **Next**.
7. On the Bind Variables page, do one of the following:
 - If the query does not reference any bind variables, click **Next** to skip Step 3.
 - To add a bind variable and work with it in the query, see [Section 5.9.1, "How to Add Bind Variables to a View Object Definition"](#).
8. On the Attribute Mappings page, click **Finish**.

Note: In the ADF Business Components wizards and editors, the default convention is to use camel-capped attribute names, beginning with a capital letter and using uppercase letters in the middle of the name to improve readability when the name comprises multiple words.

5.2.4 What Happens When You Create a Read-Only View Object

When you create a view object, JDeveloper first parses the query to infer the following from the columns in the SELECT list:

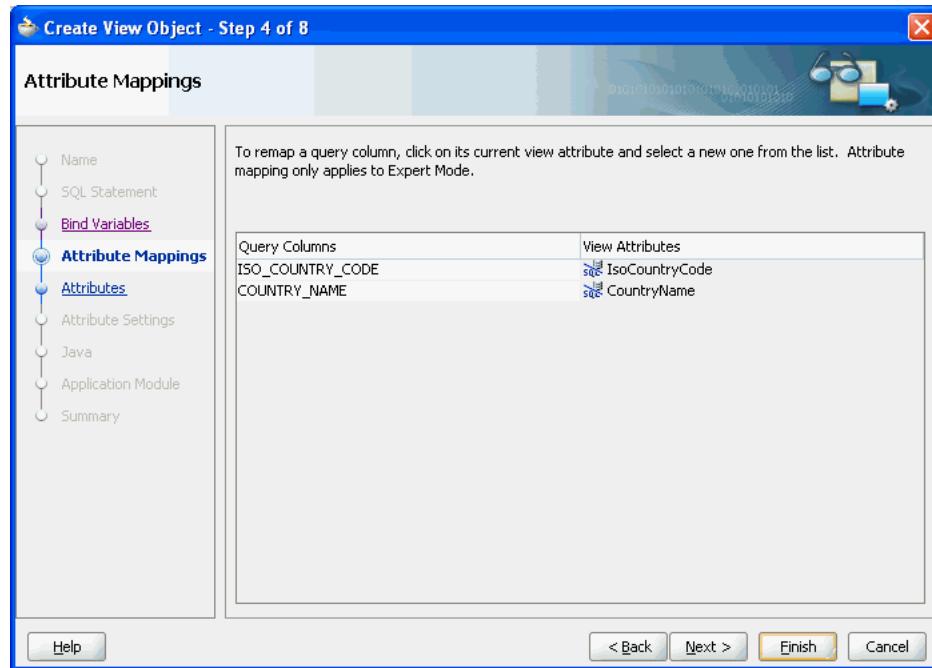
- The Java-friendly view attribute names (for example, `CountryName` instead of `COUNTRY_NAME`)

By default, the wizard creates Java-friendly view object attribute names that correspond to the SELECT list column names, as shown in [Figure 5–9](#).

For information about using view object attribute names to access the data from any row in the view object's result set by name, see [Section 6.4, "Testing View Object Instances Programmatically"](#).

- The SQL and Java data types of each attribute

Figure 5–9 Create View Object Wizard, Attribute Mappings Page



Each part of an underscore-separated column name like `SOME_COLUMN_NAME` is turned into a camel-capped word (like `SomeColumnName`) in the attribute name. While the view object attribute names correspond to the underlying query columns in the SELECT list, the attribute names at the view object level need not match necessarily.

Tip: You can rename the view object attributes to any names that might be more appropriate without changing the underlying query.

JDeveloper then creates the XML component definition file that represents the view object's declarative settings and saves it in the directory that corresponds to the name of its package. For example, the XML file created for a view object named CountriesVO in the lookups package is ./lookups/CountriesVO.xml under the project's source path.

To view the view object settings, expand the desired view object in the Application Navigator, select the XML file under the expanded view object, and open the Structure window. The Structure window displays the list of definitions, including the SQL query, the name of the entity usage, and the properties of each attribute. To open the file in the editor, double-click the corresponding .xml node.

Note: If your IDE-level Business Components Java generation preferences so indicate, the wizard may also create an optional custom view object class CountriesVOImpl.java and/or a custom view row class CountriesVORowImpl.java class.

5.2.5 How to Edit a View Object

After you've created a view object, you can edit any of its settings in the overview editor for the view object.

Performance Tip: How you configure the view object to fetch data plays a large role in the runtime performance of the view object. For information about the tuning parameters that you can edit to optimize performance, see [Section 6.3.9, "What You May Need to Know About Optimizing View Object Runtime Performance"](#).

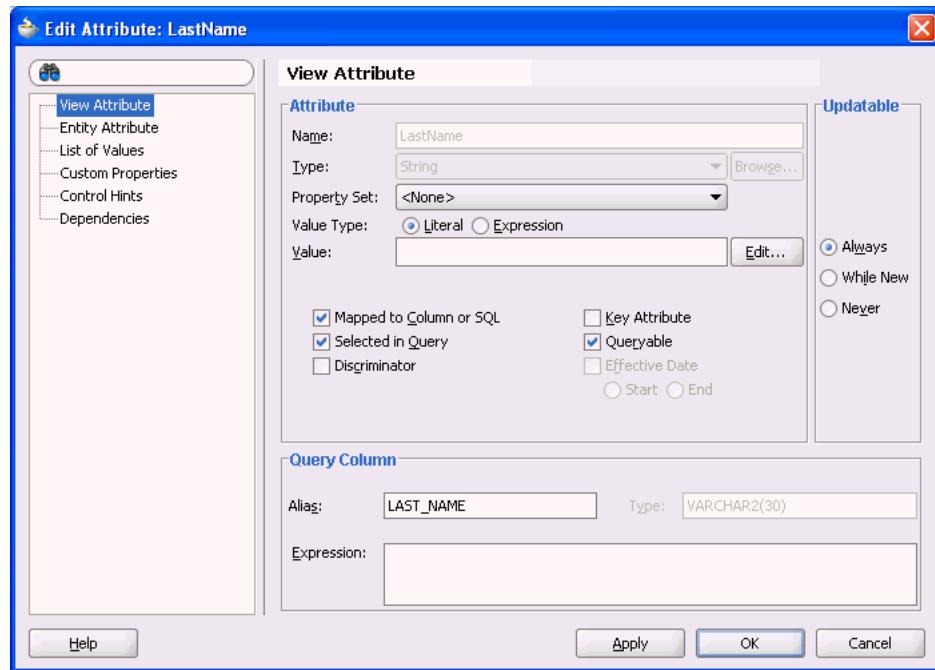
To edit a view object definition:

1. In the Application Navigator, double-click the view object to open the overview editor.
2. Select a navigation tab to open any editor page where you can adjust the SQL query, change the attribute names, add named bind variables, add UI controls hints, control Java generation options, and edit other settings.

5.2.5.1 Overriding the Inherit Properties from Underlying Entity Object Attributes

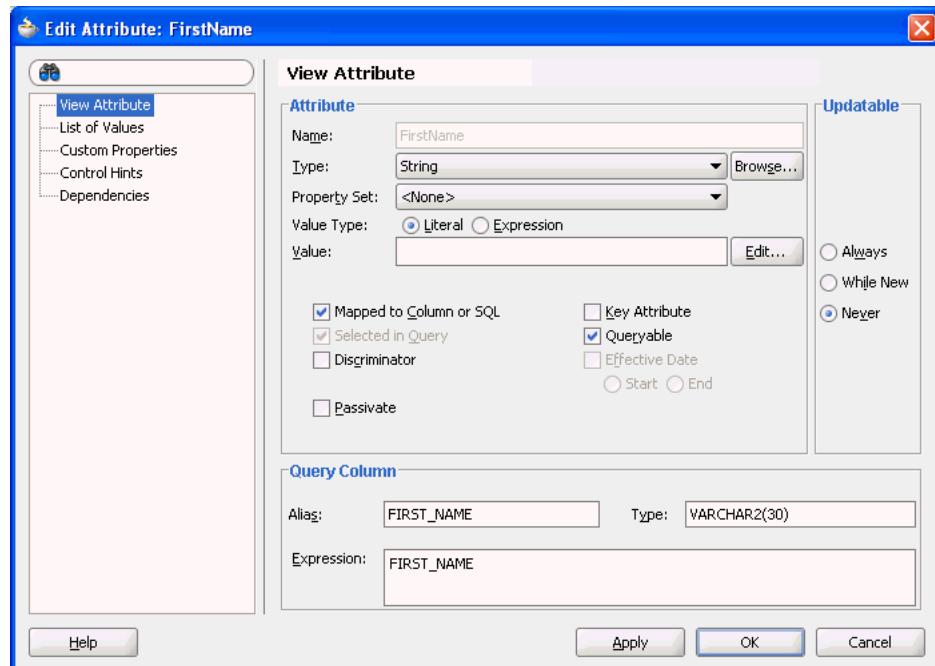
One interesting aspect of entity-based view objects is that each attribute that relates to an underlying entity object attribute inherits that attribute's properties. [Figure 5–10](#) shows the Edit Attribute dialog with the inherited attribute selected. You can see that fields like the Java attribute type and the query column type are disabled and their values are inherited from the related attribute of the underlying entity object to which this view object is related. Some properties like the attribute's data type are inherited and cannot be changed at the view object level.

Other properties like `Queryable` and `Updatable` are inherited but can be overridden as long as their overridden settings are more restrictive than the inherited settings. For example, the attribute from underlying entity object might have an `Updatable` setting of `Always`. As shown [Figure 5–10](#), the Edit Attribute dialog allows you to set the corresponding view object attribute to a more restrictive setting like `While New` or `Never`. However, if the attribute in the underlying entity object had instead an `Updatable` setting of `Never`, then the editor would not allow the view object's related attribute to have a less restrictive setting like `Always`.

Figure 5–10 View Object Attribute Properties Inherited from Underlying Entity Object

5.2.5.2 Controlling the Length, Precision, and Scale of View Object Attributes

When you display a particular attribute of the view object in the Edit Attribute dialog, you can see and change the values of the declarative settings that control its runtime behavior. One important property is the **Type** in the **Query Column** section, shown in Figure 5–11. This property records the SQL type of the column, including the length information for VARCHAR2 columns and the precision and scale information for NUMBER columns.

Figure 5–11 Custom Attribute Settings in the Edit Attribute Dialog

JDeveloper tries to infer the type of the column automatically, but for some SQL expressions the inferred value might default to VARCHAR2 (255). You can update the **Type** value for this type of attribute to reflect the correct length if you know it. In the case of read-only view objects, this property is editable in the Edit Attribute dialog you display from the overview editor for the view object. In the case of entity-based view objects, you must edit the **Type** property in the Edit Attribute dialog that you display for the entity object, as described in [Section 4.10.2, "How to Indicate Data Type Length, Precision, and Scale"](#).

For example, VARCHAR2 (30) which shows as the **Type** for the **FirstName** attribute in [Figure 5–11](#) means that it has a maximum length of 30 characters. For a NUMBER column, you would indicate a **Type** of NUMBER (7, 2) for an attribute that you want to have a precision of 7 digits and a scale of 2 digits after the decimal.

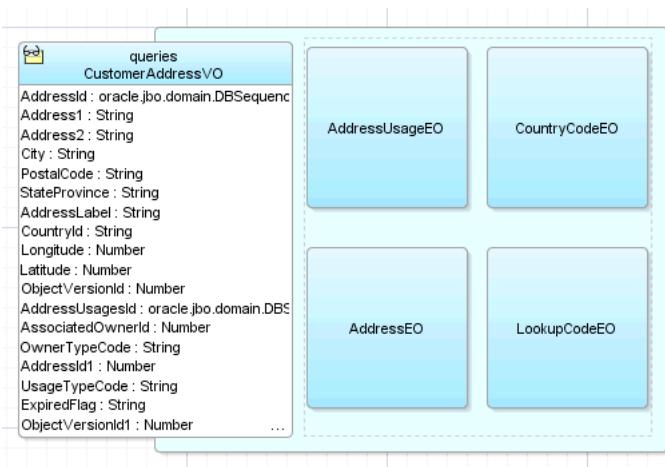
Performance Tip: Your SQL expression can control how long the describe from the database says the column is. Use the SUBSTR() function around the existing expression. For example, if you specify SUBSTR(*yourexpression*, 1, 15), then the describe from the database will inform JDeveloper that the column has a maximum length of 15 characters.

5.2.6 How to Show View Objects in a Business Components Diagram

JDeveloper's UML diagramming lets you create a Business Components diagram to visualize your business domain layer. In addition to supporting entity objects, JDeveloper's UML diagramming allows you to drop view objects onto diagrams as well to visualize their structure and entity usages. For example, if you create a new Business Components Diagram named `StoreFrontService Data Model` in the `oracle.fodemo.storefront` package, and drag the `CustomerAddressVO` view object from the Application Navigator onto the diagram, its entity usages would display, as shown in [Figure 5–12](#). When viewed as an expanded node, the diagram shows a compartment containing the view objects entity usages.

For information about creating the diagram, see [Section 4.4, "Creating an Entity Diagram for Your Business Layer"](#).

Figure 5–12 View Object and Its Entity Usages in a Business Components Diagram



5.3 Populating View Object Rows with Static Data

ADF Business Components lets you create view objects in your data model project with rows that you populate at design time. Typically, you create view objects with static data when you have a small amount of data to maintain and you do not expect that data to change frequently. The decision whether to use a lookup table from the database or whether to use a static view object based on a list of hardcoded values depends on the size and nature of the data. The static view object is useful when you have no more than 100 entries to list. Any larger number of rows should be read from the database with a conventional table-based view object. The static view object has the advantage of being easily translatable. However, all of the rows of a static view object will be retrieved at once and therefore, using no more than 100 entries yields the best performance.

Best Practice: When you need to create a view object to access a small list of static data, you should use the static view object rather than query the database. The static view object is ideal for lists not exceeding 100 rows of data. Because the Create View Object wizard saves the data in a resource message file, these data are easily translatable.

Static list view objects are useful as an LOV data source when it is not desirable to query the database to supply the list of values. Suppose your order has the following statuses: open, closed, pending. You can create a static view object with these values and define an LOV on the static view object's status attribute. Because the wizard stores the values of the status view object in a translatable resource file, the UI will display the status values using the resource file corresponding to the application's current locale.

5.3.1 How to Create Static View Objects with Data You Enter

You use the Create View Object wizard to create static view objects. The wizard lets you define the desired attributes (columns) and enter as many rows of data as necessary. The wizard displays the static data table as you create it.

Note: Because the data in a static view object does not originate in database tables, the view object will be read-only.

You can also use the Create View Object wizard to create the attributes based on data from a comma-separated value (CSV) file format like a spreadsheet file. The wizard will attempt to create the attributes that you define in the wizard with data from the first row of the flat file.

To manually create attributes for a static view object:

1. In the Application Navigator, right-click the project in which you want to create the static list view object and choose **New**.
2. In the New Gallery, expand **Business Tier**, select **ADF Business Components**, and then **View Object**, and click **OK**.
If this is the first component you're creating in the project, the Initialize Business Components Project dialog appears to allow you to select a database connection.
3. In the Initialize Business Components Project dialog, select the database connection or choose **New** to create a connection. Click **OK**.

4. In the Create View Object wizard, on the Name page, enter a package name and a view object name. Select **Rows populated at design time (Static List)** to indicate that you want to supply static list data for this view object. Click **Next**.
5. On the Attributes page, click **New** to add an attribute that corresponds to the columns in the static data table. In the New View Object Attribute dialog, enter a name and select the attribute type. Click **OK** to return to the wizard, and click **Next**.
6. On the Attribute Settings page, do nothing and click **Next**.
7. On the Static List page, click the **Add** icon to enter the data directly into the wizard page. The attributes you defined will appear as the columns for the static data table.
8. On the Java page and the Application Module pages, do nothing and click **Next**.
9. On the Summary page, click **Finish**.

5.3.2 How to Create Static View Objects with Data You Import

Using the Import feature of the Create View Object wizard, you can create a static data view object with attributes based on data from a comma-separated value (CSV) file format like a spreadsheet file. The wizard will use the first row of a CSV flat file to identify the attributes and will use the subsequent rows of the CSV file for the data for each attribute. For example, if your application needs to display choices for international currency, you might define the columns **Symbol**, **Country**, and **Description** in the first row and then add rows to define the data for each currency type, as shown in [Figure 5–13](#).

Figure 5–13 Sample Data Ready to Import from CSV Flat File

	A	B	C
1	Symbol	Country	Description
2	USD	United States of America	Dollars
3	CNY	P.R. China	Yuan Renminbi
4	EUR	Europe	Euro
5	JPY	Japan	Yen

To create attributes of a static view object based on a flat file:

1. In the Application Navigator, right-click the project in which you want to create the static list view object and choose **New**.
2. In the New Gallery, expand **Business Tier**, select **ADF Business Components**, and then **View Object**, and click **OK**.
If this is the first component you're creating in the project, the Initialize Business Components Project dialog appears to allow you to select a database connection.
3. In the Initialize Business Components Project dialog, select the database connection or choose **New** to create a connection. Click **OK**.
4. In the Create View Object wizard, on the Name page, enter a package name and a view object name. Select **Rows populated at design time (Static List)** to indicate that you want to supply static list data for this view object. Click **Next**.
5. On the Attributes page, optionally, click **New** to add an attribute that corresponds to the columns in the static data table. In the New View Object Attribute dialog, enter a name and select the attribute type. Click **OK** to return to the wizard, and click **Next**.

When the static data will be loaded from a CSV flat file, you can optionally skip this step. If you do not create the attributes yourself, the wizard will attempt to use the first row of the CSV file to create the attributes. However, if you create the attributes in the wizard, then the attributes you create must match the order of the columns defined by the flat file. If you have created fewer attributes than columns, the wizard will ignore extra columns during import. Conversely, if you create more attributes than columns, the wizard will define extra attributes with the value NULL.

6. On the Attribute Settings page, do nothing and click **Next**.
7. On the Static List page, click **Import** to locate the CSV file and display the data in the wizard. Verify the data and edit the values as needed.
To edit an attribute value, double-click in the value field.
8. Optionally, click the **Add** icon or **Remove** icon to change the number of rows of data. Click **Next**.
To enter values for the attributes of a new row, double-click in the value field.
9. On the Application Module page, do nothing and click **Next**.
10. On the Summary page, click **Finish**.

5.3.3 What Happens When You Create a Static List View Object

When you create a static view object, the overview editor for the view object displays the rows of data that you defined in the wizard. You can use the editor to define additional data, as shown in [Figure 5–14](#).

Figure 5–14 Static Values Page Displays Data

The screenshot shows the 'Static Values' tab selected in the left sidebar of the editor. The main area displays a table with the following data:

Symbol	Country	Description
USD	United States of America	Dollars
CNY	P.R. China	Yuan Renminbi
EUR	Europe	Euro
JPY	Japan	Yen

The generated XML definition for the static view object contains one transient attribute for each column of data. For example, if you import a CSV file with data that describes international currency, your static view object might contain a transient attribute for `Symbol`, `Country`, and `Description`, as shown in [Example 5–1](#).

Example 5–1 XML Definition for Static View Object

```
<ViewObject
...
// Transient attribute for first column
<ViewAttribute
  Name="Symbol"
  IsUpdateable="false"
  IsSelected="false"
  IsPersistent="false"
  PrecisionRule="true"
  Precision="255"
  Type="java.lang.String"
```

```

ColumnType="VARCHAR2"
AliasName="Symbol"
SQLType="VARCHAR"/>
// Transient attribute for second column
<ViewAttribute
  Name="Country"
  IsUpdateable="false"
  IsPersistent="false"
  PrecisionRule="true"
  Precision="255"
  Type="java.lang.String"
  ColumnType="VARCHAR"
  AliasName="Country"
  SQLType="VARCHAR"/>
// Transient attribute for third column
<ViewAttribute
  Name="Description"
  IsUpdateable="false"
  IsPersistent="false"
  PrecisionRule="true"
  Precision="255"
  Type="java.lang.String"
  ColumnType="VARCHAR"
  AliasName="Description"
  SQLType="VARCHAR"/>
<StaticList
  Rows="4"
  Columns="3"/>
// Reference to file that contains static data
<ResourceBundle>
  <PropertiesBundle
    PropertiesFile="model.ModelBundle"/>
</ResourceBundle>
</ViewObject>

```

Because the data is static, the overview editor displays no Query page and the generated XML definition for the static view object contains no query statement. Instead, the <ResourceBundle> element in the XML definition references a resource bundle file. The resource bundle file describes the rows of data and also lets you localize the data. When the default resource bundle type is used, the file `ModelNameBundle.properties` appears in the data model project, as shown in [Example 5–2](#).

Example 5–2 Default Resource Bundle File for Static View Object

```

model.ViewObj.SL_0_0=USD
model.ViewObj.SL_0_1=United States of America
model.ViewObj.SL_0_2=Dollars
model.ViewObj.SL_1_0=CNY
model.ViewObj.SL_1_1=P.R. China
model.ViewObj.SL_1_2=Yuan Renminbi
model.ViewObj.SL_2_0=EUR
model.ViewObj.SL_2_1=Europe
model.ViewObj.SL_2_2=Euro
model.ViewObj.SL_3_0=JPY
model.ViewObj.SL_3_1=Japan
model.ViewObj.SL_3_2=Yen

```

5.3.4 Editing Static List View Objects

When you need to make changes to the static list table, double-click the view object in the Application Navigator to open the overview editor for the view object. You can add and delete attributes (columns in the static list table), add or delete rows (data in the static list table), sort individual rows, and modify individual attribute values. The editor will update the view object definition file and save the attribute names in the message bundle file.

5.3.5 What You May Need to Know About Static List View Objects

The static list view object has a limited purpose in the application module's data model. Unlike entity-based view objects, static list view objects will not be updatable. You use the static list view object when you want to display read-only data to the end user and you do not want to create a database table for the small amount of data the static list table contains.

5.4 Limiting View Object Rows Using Effective Date Ranges

Applications that need to query data over a specific date range can generate *date-effective row sets*. To define an date-effective view object you must create an entity-based view object that is based on an date-effective entity object. User control over the view object's effective date usage is supported by metadata on the view object at design time. At runtime, ADF Business Components generates the query filter that will limit the view rows to an effective date.

5.4.1 How to Create an Date-Effective View Object

Whether or not the query filter for an effective date will be generated depends on the value of the **Effective Dated** property displayed in the Property Inspector for the view object (to view the property, select any tab in the overview editor for the view object other than **Attributes**).

Note: Because the date-effective view object must be based on an date-effective entity object, setting a view object's **Effective Dated** property to **True** without an underlying date-effective entity object, will result in a runtime exception.

The overview editor for the view object does not display the date-effective query clause in the WHERE clause. You can use the Explain Plan dialog or Test Query dialog to view the clause. A typical query filter for effective dates looks like this:

```
(:Bind_SysEffectiveDate BETWEEN Person.EFFECTIVE_START_DATE AND  
Person.EFFECTIVE_END_DATE)
```

At runtime, the bind value for the query is obtained from a property of the root application module. In order to set the effective date for a transaction, use code similar to the following snippet:

```
am.setProperty(ApplicationModule.EFF_DT_PROPERTY_STR, new  
Date("2008-10-01));
```

If you do not set EFF_DT_PROPERTY_STR on the application module, the current date is used in the query filter, and the view object returns the effective rows filtered by the current date.

The view object has its own transient attribute, `SysEffectiveDate`, that you can use to set the effective date for view rows. Otherwise, the `SysEffectiveDate` attribute value for new rows and defaulted rows is derived from the application module. ADF Business Components propagates the effective date from the view row to the entity object during DML operations only.

To enable effective dates for view object using the `SysEffectiveDate` attribute:

1. Create an effective dated entity object, as described in [Section 4.2.8, "How to Store Data Pertaining to a Specific Point in Time"](#).
2. Use the Create View Object wizard to create the view object based on the effective dated entity object.

In the Attributes page of the wizard, be sure to add the date-effective attributes that specify the Start Date and End Date on the entity object to the **Selected** list for the view object.

3. Open the newly created view object in the overview editor and click the General tab.
4. In the Property Inspector, expand the **Name** category.

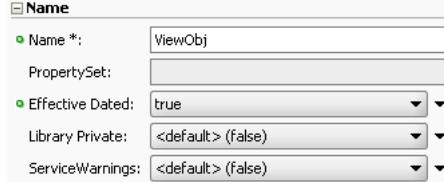
If necessary, choose **Property Inspector** from the **View** menu to display the Property Inspector.

If the **Name** category is not displayed in the Property Inspector, click the **General** tab in the overview editor to set the proper focus.

5. Verify that the context menu for the **Effective Dated** property displays **True**.

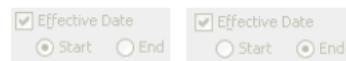
The date-effective view object inherits the **Effective Dated** property setting from the date-effective entity object, as shown in [Figure 5-15](#).

Figure 5-15 Property Inspector Displays Effective Dated Property Setting



6. In the Attributes page of the overview editor, double-click the attribute for the start date. In the Edit Attribute dialog verify that **Effective Date** is enabled and that **Start** is selected, as shown in [Figure 5-16](#). Verify that the attribute for the end date is also enabled correctly as shown in [Figure 5-16](#).

Figure 5-16 Edit Attribute Dialog Displays Effective Date Settings



7. No additional steps are required once you have confirmed that the view object has inherited the desired attributes from the date-effective entity object.

5.4.2 How to Create New View Rows Using Date-Effective View Objects

Creating (inserting) date-effective rows is similar to creating or inserting ordinary view rows. The start date and end date can be specified as follows:

- The user specifies the effective date on the application module. The start date is set to the effective date, and the end date is set to end of time.
- The user specifies values for the start date and the end date (advanced).

In either case, during entity validation, the new row is checked to ensure that it does not introduce any gaps or overlaps. During post time, ADF Business Components will acquire a lock on the previous row to ensure that the gap or overlaps are not created upon the row insert.

5.4.3 How to Update Date-Effective View Rows

You can update view rows by using API calls like `Row.setAttribute()`. ADF Business Components supports various modes to initiate the row update.

To set the update mode, invoke the `Row.setEffectiveDateMode(int mode)` method with one of the following mode constants.

- **CORRECTION** (Correction Mode)
The effective start date and effective end dates remain unchanged. The values of the other attributes may change. This is the standard row update behavior.
- **UPDATE** (Update Mode)
The effective end date of the row will be set to the effective date. All user modifications to the row values are reverted on this row. A new row with the modified values is created. The effective start date of the new row is set to the effective date plus one day, and the effective end date is set to end of time. The new row will appear after the transaction is posted to the database.
- **UPDATE_OVERRIDE** (Update Override Mode)
The effective end date of the modified row will be set to the effective date. The effective start date of the next row is set to effective date plus one day.
- **UPDATE_CHANGE_INSERT** (Change Insert Mode)
The effective end date of the modified row should be set to the effective date. All user modifications to the row values are reverted on this row. A new row with the modified values will be created. The effective start date of the new row is set to effective date plus one day, and the effective end date is set to effective start date of the next row minus one day. The new row will appear after the transaction is posted to the database.

5.4.4 How to Delete Date-Effective View Rows

ADF Business Components supports various modes to initiate the row deletion. You can mark view rows for deletion by using API calls like `RowSet.remove.CurrentRow()` or `Row.remove()`.

To set the deletion mode, invoke the `Row.setEffectiveDateMode(int mode)` method with one of the following mode constants.

- **DELETE** (Delete Mode)
The effective date of the row is set to the effective date. The operation for this row is changed from delete to update. All rows with the same non-effective date key values and with an effective start date greater than the effective date are deleted.
- **DELETE_NEXT_CHANGE** (Delete Next Change Mode)

The effective end date of the row is set to the effective end date of the next row with the same noneffective date key values. The operation for this row is changed from delete to update. The next row is deleted.

- **FUTURE_CHANGE** (Delete Future Change Mode)

The effective end date of the row is set to the end of time. The operation for this row is changed from delete to update. All future rows with the same noneffective date key values are deleted.

- **ZAP** (Zap Mode)

All rows with the same non-effective date key values are deleted.

The effective date mode constants are defined on the row interface as well.

5.4.5 What Happens When You Create a Date-Effective View Object

When you create an date-effective view object, the view object inherits the transient attribute `SysEffectiveDate` from the entity object to store the effective date for the row. Typically, the insert/update/delete operations modify the transient attribute while Oracle ADF decides the appropriate values for effective start date and effective end date.

The query displayed in the overview editor for the date-effective view object does not display the WHERE clause needed to filter the effective date range. To view the full query for the date-effective view object, including the WHERE clause, edit the query and click **Explain Plan** in the Edit Query dialog. The following sample shows a typical query and query filter for effective dates:

```
SELECT OrdersVO.ORDER_ID,
       OrdersVO.CREATION_DATE,
       OrdersVO.LAST_UPDATE_DATE
  FROM ORDERS OrdersVO
 WHERE (:Bind_SysEffectiveDate BETWEEN OrdersVO.CREATION_DATE AND
          OrdersVO.LAST_UPDATE_DATE)
```

Example 5–3 shows sample XML entries that are generated when you create an date-effective view object.

Example 5–3 XML Definition for Date-Effective View Object

```
<ViewObject
  ...
  // Property that enables date-effective view object.
  IsEffectiveDated="true">
  <EntityUsage
    Name="Orders1"
    Entity="model.OrdersDatedEO"
    JoinType="INNER JOIN"/>
  // Attribute identified as the start date
  <ViewAttribute
    Name="CreationDate"
    IsNotNull="true"
    PrecisionRule="true"
    IsEffectiveStartDate="true"
    EntityAttrName="CreationDate"
    EntityUsage="Orders1"
    AliasName="CREATION_DATE"/>
  // Attribute identified as the end date
  <ViewAttribute
```

```
Name="LastUpdateDate"
IsNotNull="true"
PrecisionRule="true"
IsEffectiveEndDate="true"
EntityAttrName="LastUpdateDate"
EntityUsage="Orders1"
AliasName="LAST_UPDATE_DATE" />
// The SysEffectiveDate transient attribute
<ViewAttribute
    Name="SysEffectiveDate"
    IsPersistent="false"
    PrecisionRule="true"
    Type="oracle.jbo.domain.Date"
    ColumnType="VARCHAR2"
    AliasName="SysEffectiveDate"
    Passivate="true"
    SQLType="DATE" />
</ViewObject>
```

5.4.6 What You May Need to Know About Date-Effective View Objects and View Links

Effective dated associations and view links allow queries to be generated that take the effective date into account. The effective date of the driving row is passed in as a bind parameter during the query execution.

While it is possible to create a noneffective dated association between two entities when using the Create Association wizard or Create View Link wizard, JDeveloper will by default make the association or link effective dated if one of the ends is effective dated. However, when the association or view link exists between an effective dated and a noneffective dated object, then at runtime ADF Business Components will inspect the effective dated nature of the view object or entity object before generating the query clause and binding the effective date. The effective date is first obtained from the driving row. If it is not available, then it is obtained from the property EFF_DT_PROPERTY_STR of the root application module. If you do not set EFF_DT_PROPERTY_STR for the application module, the current date is used in the query filter on the driving row and applied to the other side of the association or view link.

5.5 Working with Multiple Tables in Join Query Results

Many queries you will work with will involve multiple tables that are related by foreign keys. In this scenario, you join the tables in a single view object query to show additional descriptive information in each row of the main query result. You use the Create View Object wizard to define the query using declarative options. Whether your view object is read-only or entity-based determines how you can define the join:

- When you work with entity-based view objects, the Create View Object wizard uses an existing association defined between the entities to automatically build the view object's join WHERE clause. You can declaratively specify the type of join you want to result from the entity objects. Inner join (equijoin) and outer joins are both supported.
- When you work with read-only view objects, you will use the SQL Builder dialog to build the view object's join WHERE clause. In this case, you must select the columns from the tables that you want to join.

Figure 5-17 illustrates the rows resulting from two tables queried by a view object that defines a join query. The join is a single flattened result.

Figure 5–17 Join Query Result

**Join
Query**

101	Order	322	Pat	Fay
212	Order	322	Pat	Fay
222	Order	310	John	Chen
123	Order	300	Steven	King

 Persons
 Orders

5.5.1 How to Create Joins for Entity-Based View Objects

It is extremely common in business applications to supplement information from a *primary* business domain object with secondary reference information to help the end user understand what foreign key attributes represent. Take the example of the OrderItems entity object. It contains foreign key attribute of type Number like:

- ProductId, representing the product to which the order item pertains

From experience, you know that showing an end user exclusively these "raw" numerical values won't be very helpful. Ideally, reference information from the view object's related entity objects should be displayed to improve the application's usability. One typical solution involves performing a join query that retrieves the combination of the primary and reference information. This is equivalent to populating "dummy" fields in each queried row with reference information based on extra queries against the lookup tables.

When the end user can *change* the foreign key values by editing the data, this presents an additional challenge. Luckily, entity-based view objects support easily including reference information that's always up to date. The key requirement to leverage this feature is the presence of associations between the entity object that act as the view object's primary entity usage and the entity objects that contribute reference information.

To include reference entities in a join view object, use the Create View Object wizard. The Create View Object wizard lets you specify the type of join:

- **Inner Join**

Select when you want the view object to return all rows between two or more entity objects, where each entity defines the same primary key column. The inner join view object will not return rows when a primary key value is missing from the joined entities.

- **Outer Join**

Select when you want the view object to return all rows that exist in one entity object, even though corresponding rows do not exist in the joined entity object. Both left and right outer join types are supported. In the left outer join, you will return only the non-NULL rows of the first entity object (primary) in the Selected list, but the returned rows of subsequent participant entities (secondary) may be NULL. The right outer join specifies the reverse scenario: you will return only the non-NULL rows of the subsequent participant entity objects (secondary) in the Selected list, but the returned rows of first entity object (primary) in the list may be NULL.

Both inner joins and outer joins are supported with the following options:

- **Reference**

Select when you want the data from the entity object to be treated as reference information for the view object. Automatic lookup of the data is supported and attribute values will be dynamically fetched from the entity cache when a controlling key attribute changes.

- **Updatable**

Deselect when you want to prevent the view object from modifying any entity attributes in the entity object. By default, the first entity object (primary) in the **Selected** list is updatable and subsequent entity objects (secondary) are not updatable. To understand how to create a join view object with *multiple* updatable entity usages, see [Section 35.10, "Creating a View Object with Multiple Updatable Entities"](#).

- **Participate in row delete**

Select when you have defined the entity as updatable and you want the action of removing rows in the UI to delete the participating reference entity object. This option is disabled for the primary entity. For example, while it may be possible to delete an order item, it should not be possible to delete the order when a remove row is called from the join view object.

To create a view object that joins entity objects:

1. In the Application Navigator, right-click the project in which you want to create the view object and choose **New**.

When you want to modify an existing view object that you created to include reference information from its related entity objects, double-click the view object and open the Entity Objects page in the overview editor for the view object.

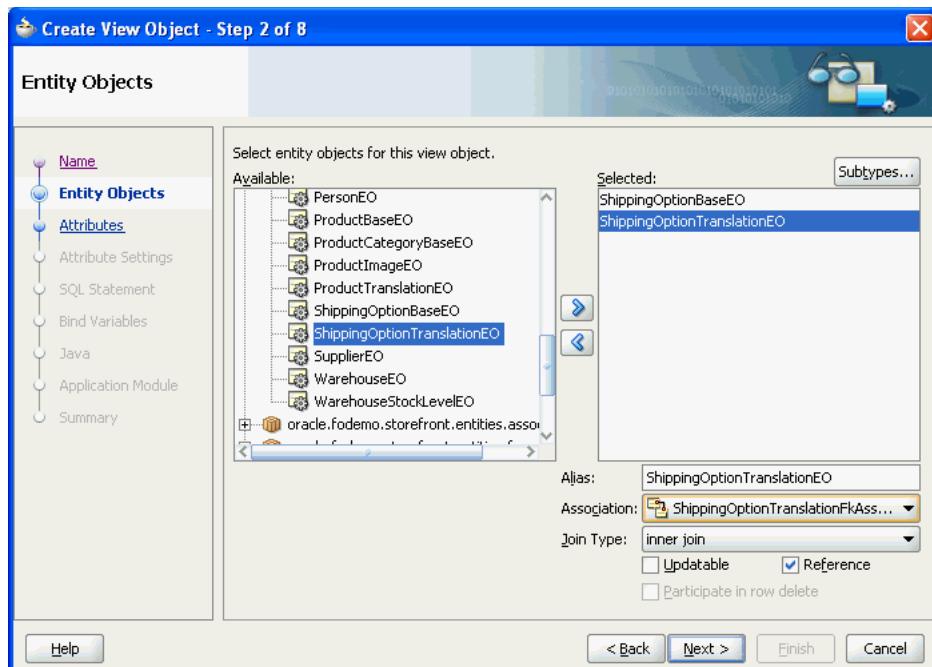
2. In the New Gallery, expand **Business Tier**, select **ADF Business Components**, then **View Object**, and click **OK**.
3. In the Create View Object wizard, on the Name page, enter a package name and a view object name. Keep the default setting **Updatable Access Through Entity Objects** enabled to indicate that you want this view object to manage data with its base entity object. Click **Next**.
4. In the Entity Objects page, the first entity usage in the **Selected** list is known as the *primary* entity usage for the view object. Select the primary entity object from the **Available** list and shuttle it to the **Selected** list.

The list is not limited to a single, primary entity usage.

5. To add additional, secondary entity objects to the view object, select them in the **Available** list and shuttle them to the **Selected** list.

The **Association** dropdown list shows you the name of the association that relates the selected secondary entity usage to the primary one. For example, [Figure 5–18](#) shows the result of adding one secondary reference entity usage, `ShippingOptionTranslationEO`, in addition to the primary `ShippingOptionBaseEO` entity usage. The association that relates to this secondary entity usage is `ShippingOptionTranslationFkAssociation`.

Figure 5–18 Create View Object Wizard, Entity Objects Page



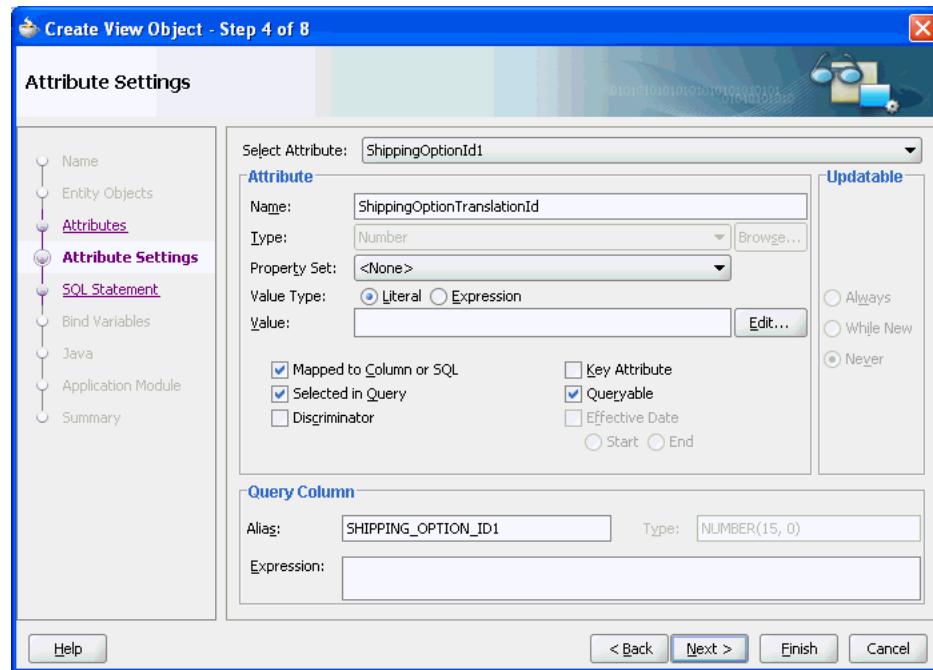
6. Optionally, use the **Alias** field to give a more meaningful name to the entity usage when the default name is not clear.
7. If you add multiple entity usages for the same entity, use the **Association** dropdown list to select which association represents that usage's relationship to the primary entity usage. Click **Next**.

For each secondary entity usage, the **Reference** option is enabled to indicate that the entity provides reference information and that it is not the primary entity. The **Updatable** option is disabled. This combination represents the typical usage. However, when you want to create a join view object with multiple, updatable entity usages, see [Section 35.10, "Creating a View Object with Multiple Updatable Entities"](#).

Secondary entity usages that are updatable can also have the **Participate in row delete** option enabled. This will allow secondary entity attributes to appear NULL when the primary entity is displayed.

8. On the Attributes page, select the attributes you want each entity object usage to contribute to the view object. Click **Next**.
9. On the Attribute Settings page, you can rename an attribute when the names are not as clear as they ought to be.

The same attribute name often results when the reference and secondary entity objects derive from the same table. [Figure 5–19](#) shows the attribute `ShippingOptionId1` in the **Select Attribute** dropdown list, which has been renamed to `ShippingOptionTranslationId` in the **Name** field.

Figure 5–19 Create View Object Wizard, Attribute Settings Page

10. Click **Finish**.

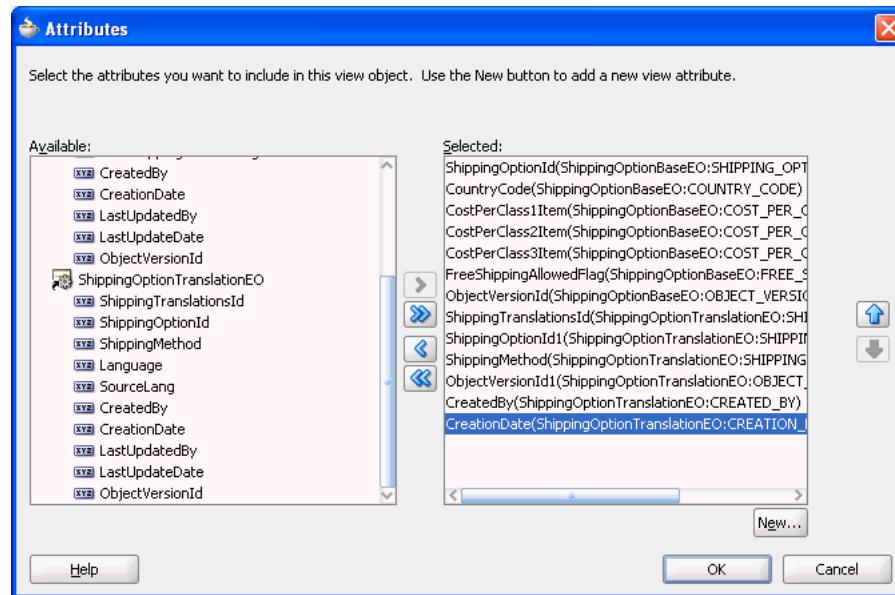
5.5.2 How to Select Additional Attributes from Reference Entity Usages

After adding secondary entity usages, you can use the overview editor for the view object to select the specific, additional *attributes* from these new usages that you want to include in the view object. To edit the list of attributes from reference entity usages, in the Attribute page of the view object's overview editor, click the **Add from Entity** button and use the Attributes dialog to customize the selected attributes list.

Figure 5–20 illustrates the result of shuttling the following extra attributes into the **Selected** list:

- The `CreatedBy` attribute from the `ShippingOptionTranslationEO` entity usage
- The `CreationDate` attribute from the `ShippingOptionTranslationEO` entity usage

Figure 5–20 Selecting Additional Reference Entity Attributes to Include in the View Object



Note that even if you didn't intend to include them, JDeveloper automatically verifies that the primary key attribute from each entity usage is part of the **Selected** list. If it's not already present in the list, JDeveloper adds it for you. When you are finished, the overview editor Query page shows that JDeveloper has included the new columns in the SELECT statement.

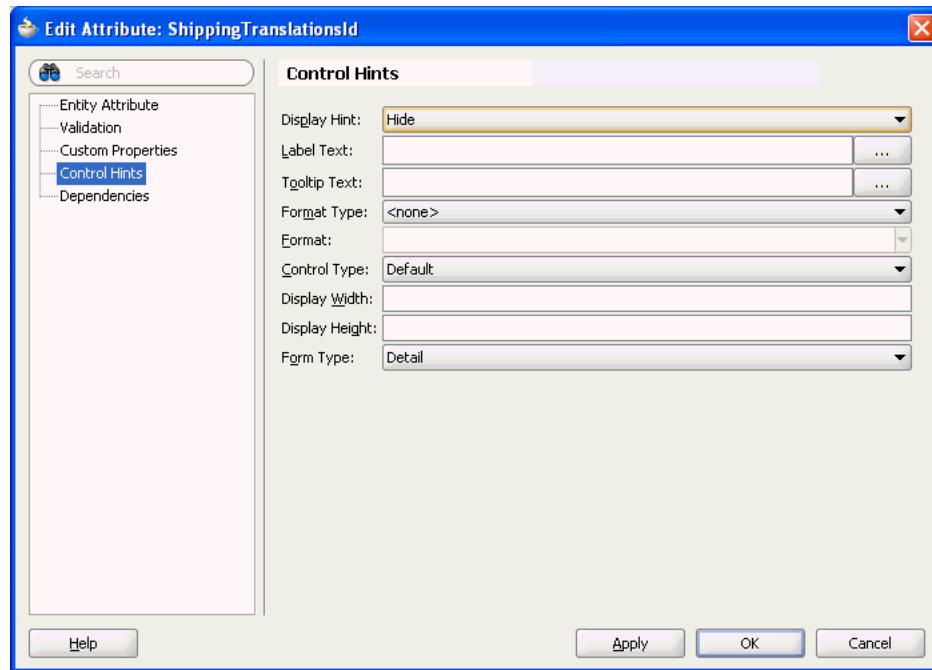
5.5.3 How to Remove Unnecessary Key Attributes from Reference Entity Usages

The view object attribute corresponding to the primary key attribute of the primary entity usage acts as the primary key for identifying the view row. When you add secondary entity usages, JDeveloper marks the view object attributes corresponding to their primary key attributes as part of the view row key as well. When your view object consists of a single updatable primary entity usage and a number of reference entity usages, the primary key attribute from the primary entity usage is enough to uniquely identify the view row. Further key attributes contributed by secondary entity usages are not necessary and you should disable their **Key Attribute** settings.

For example, based on the view object with primary entity usage `ShippingOptionEO`, you could disable the **Key Attribute** property for the `ShippingOptionTranslationEO` entity usage so that this property is no longer selected for this additional key attribute: `ShippingTranslationsId`. To edit the key attribute, select the attribute in the Attributes page of the view object's overview editor and click the **Edit** icon. You can set the attribute's **Key Attribute** property in the Entity Attributes page of the Edit Attribute dialog.

5.5.4 How to Hide the Primary Key Attributes from Reference Entity Usages

Since you generally won't want to display the primary key attributes that were automatically added to the view object, you can set the attribute's **Display Hint** property in the Control Hints page of the Edit Attribute dialog to **Hide**, as shown in Figure 5–21. To edit the attribute's control hints, select the attribute in the Attributes page of the view object's overview editor and click the **Edit** icon.

Figure 5–21 Primary Key Attribute Hidden from Reference Entity Usages

5.5.5 How to Modify a Default Join Clause to Be Outer Join When Appropriate

When JDeveloper creates the WHERE clause for the join between the table for the primary entity usage and the tables for secondary entity usages related to it, by default it always creates inner joins. You can modify the default inner join clause to be an outer join when appropriate.

For example, assume that a translation shipping option is not yet assigned to a base shipping option. In this case, the AssignedTo attribute will be NULL. The default inner join condition will not retrieve these unassigned service requests. Assuming that you want unassigned shipping options to be viewable and updatable through the ShippingOptionsVO view object, you can use the Entity Object page of the overview editor for the view object to change the query into an left outer join to the USER table for the possibly null ASSIGNED_TO column value. When you add the Technician entity to the view object, you would select the **left outer join** as the join type. The updated WHERE clause shown below includes the additional (+) operator on the right side of the equals sign for the related table whose data is allowed to be missing in the left outer join:

```
ShippingOptionBaseEO.SHIPPING_OPTION_ID =
    ShippingOptionTranslationEO.SHIPPING_OPTION_ID(+)
```

5.5.6 What Happens When You Reference Entities in a View Object

When you create a join view object to include secondary entity usages by reference, JDeveloper updates the view object's XML component definition to include information about the additional entity usages. For example, the ShippingOptionsVO.xml file for the view object includes an additional reference entity usage. You will see this information recorded in the multiple `<EntityUsage>` elements. For example, [Example 5–4](#) shows an entity usage entry that defines the primary entity usage.

Example 5–4 Primary Entity Usage

```
<EntityUsage
    Name="ShippingOptionBaseEO"
    Entity="oracle.fodemo.storefront.entities.ShippingOptionBaseEO" />
```

The secondary reference entity usages will have a slightly different entry, including information about the association that relates it to the primary entity usage, like the entity usage shown in [Example 5–5](#).

Example 5–5 Secondary Reference Entity Usage

```
<EntityUsage
    Name="ShippingOptionTranslationEO"
    Entity="oracle.fodemo.storefront.entities.ShippingOptionTranslationEO"
    Association="oracle.fodemo.storefront.entities.associations.
        ShippingOptionTranslationFkAssoc"
    AssociationEnd="oracle.fodemo.storefront.entities.associations.
        ShippingOptionTranslationFkAssoc.ShippingOptionTranslation"
    SourceUsage="oracle.fodemo.storefront.store.queries.ShippingOptionsVO.
        ShippingOptionBaseEO"
    ReadOnly="true"
    Reference="true" />
```

Each attribute entry in the XML file indicates which entity usage it references. For example, the entry for the `ShippingOptionId` attribute in [Example 5–6](#) shows that it's related to the `ShippingOptionBaseEO` entity usage, while the `ShippingMethod` attribute is related to the `ShippingOptionTranslationEO` entity usage.

Example 5–6 Entity Usage Reference of View Object Attribute

```
<ViewAttribute
    Name="ShippingOptionId"
    IsNotNull="true"
    EntityAttrName="ShippingOptionId"
    EntityUsage="ShippingOptionBaseEO"
    AliasName="SHIPPING_OPTION_ID" >
</ViewAttribute>
...
<ViewAttribute
    Name="ShippingMethod"
    IsUpdatable="true"
    IsNotNull="true"
    EntityAttrName="ShippingMethod"
    EntityUsage="ShippingOptionTranslationEO"
    AliasName="SHIPPING_METHOD" >
</ViewAttribute>
```

The Create View Object wizard uses this association information at design time to automatically build the view object's join `WHERE` clause. It uses the information at runtime to enable keeping the reference information up to date when the end user changes foreign key attribute values.

5.5.7 How to Create Joins for Read-Only View Objects

To create a read-only view object joining two tables, use the Create View Object wizard.

To create a read-only view object joining two tables:

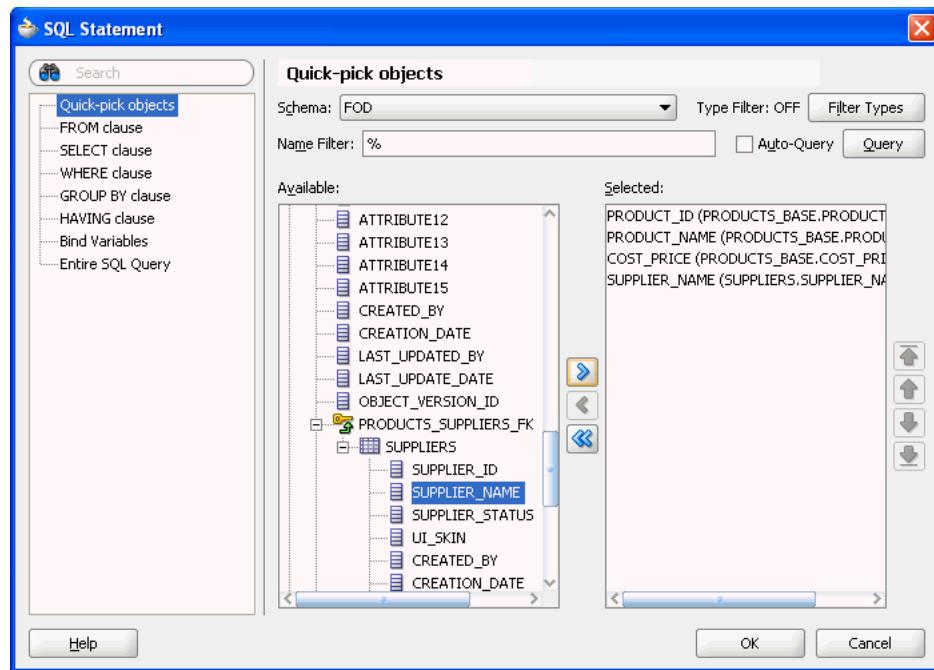
1. In the Application Navigator, right-click the project in which you want to create the view object and choose **New**.
2. In the New Gallery, expand **Business Tier**, select **ADF Business Components**, then **View Object**, and click **OK**.
3. In the Initialize Business Components Project dialog, select the database connection or choose **New** to create a connection. Click **OK**.
4. In the Create View Object wizard, on the Name page, enter a package name and a view object name. Select **Read-only Access** to indicate that you want this view object to manage data with read-only access. Click **Next**.
5. On the Query page, use one of the following techniques to create the SQL query statement that joins the desired tables:
 - Paste any valid SQL statement into the **Query Statement** box.
 - Click **Query Builder** to open the SQL Statement dialog and use the interactive query builder, as described in [Section 5.5.9, "How to Use the Query Builder with Read-Only View Objects"](#).
6. After entering or building the query statement, click **Next**.
7. On the Bind Variables page, do one of the following:
 - If the query does not reference any bind variables, click **Next** to skip Step 3.
 - To add a bind variable and work with it in the query, see [Section 5.9.1, "How to Add Bind Variables to a View Object Definition"](#).
8. On the Attribute Mappings page, click **Finish**.

5.5.8 How to Test the Join View

To test the new view object, edit the application module and on the Data Model page add an instance of the new view object to the data model. Then, use the Business Component Browser to verify that the join query is working as expected. For details about editing the data model and running the Business Component Browser, see [Section 6.3, "Testing View Object Instances Using the Business Component Browser"](#).

5.5.9 How to Use the Query Builder with Read-Only View Objects

The Quick-pick objects page of the SQL Statement dialog lets you view the tables in your schema, including the foreign keys that relate them to other tables. To include columns in the select list of the query, shuttle the desired columns from the **Available** list to the **Selected** list. For example, [Figure 5–22](#) shows the result of selecting the PRODUCT_ID, PRODUCT_NAME, and COST_PRICE columns from the PRODUCTS table, along with the SUPPLIER_NAME column from the SUPPLIERS table. The column from the second table appears, beneath the PRODUCTS_SUPPLIERS_FK foreign key in the **Available** list. When you select columns from tables joined by a foreign key, the query builder automatically determines the required join clause for you.

Figure 5–22 View Object Query Builder to Define a Join

Optionally, use the WHERE clause page of the SQL Statement dialog to define the expression. To finish creating the query, click **OK** in the SQL Statement dialog. The Edit Query dialog will show a query like the one shown in [Example 5–7](#).

Example 5–7 Creating a Query Using SQL Builder

```
SELECT
  PRODUCTS_BASE.PRODUCT_ID PRODUCT_ID,
  PRODUCTS_BASE.PRODUCT_NAME PRODUCT_NAME,
  PRODUCTS_BASE.COST_PRICE COST_PRICE,
  SUPPLIERS.SUPPLIER_NAME SUPPLIER_NAME
FROM
  PRODUCTS_BASE JOIN SUPPLIERS USING (SUPPLIER_ID)
```

You can use the Attributes page of the Create View Object wizard to rename the view object attribute directly as part of the creation process. Renaming the view object here saves you from having to edit the view object again, when you already know the attribute names that you'd like to use. As an alternative, you can also alter the default Java-friendly name of the view object attributes by assigning a column alias, as described in [Section 5.8.2, "How to Name Attributes in Expert Mode"](#).

5.5.10 What You May Need to Know About Join View Objects

If your view objects reference multiple entity objects, they are displayed as separate entity usages on a business components diagram.

5.6 Working with Multiple Tables in a Master-Detail Hierarchy

Many queries you will work with will involve multiple tables that are related by foreign keys. In this scenario, you can create separate view objects that query the related information and then link a "source" view object to one or more "target" view objects to form a master-detail hierarchy.

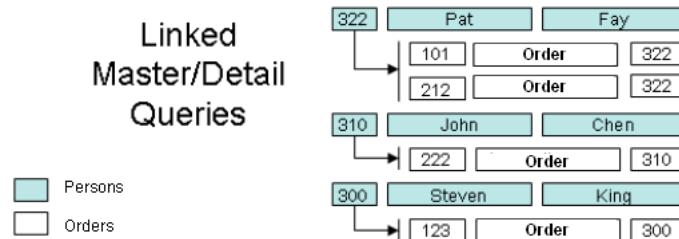
There are two ways you might handle this situation. You can either:

- Create a view link that defines how the source and target view objects relate.
- Create a view link based on an association between entity objects when the source and target view objects are based on the underlying entity objects' association.

In either case, you use the Create View Link wizard to define the relationship.

[Figure 5–23](#) illustrates the multilevel result that master-detail linked queries produce.

Figure 5–23 Linked Master-Detail Queries



5.6.1 How to Create a Master-Detail Hierarchy for Read-Only View Objects

When you want to show the user a set of master rows, and for each master row a set of coordinated detail rows, then you can create view links to define how you want the master and detail view objects to relate. For example, you could link the `PersonsVO` view object to the `Orders` view object to create a master-detail hierarchy of customers and the related set of orders they have placed.

To create the view link, use the Create View Link wizard.

To create a view link between read-only view objects:

1. In the Application Navigator, right-click the project in which you want to create the view object and choose **New**.
2. In the New Gallery, expand **Business Tier**, select **ADF Business Components**, then **View Link**, and click **OK**.
3. In the Create View Link wizard, on the Name page, enter a package name and a view link name. For example, given the purpose of the view link, a name like `OrdersPlacedBy` is a valid name. Click **Next**.
4. On the View Objects page, select a "source" attribute from the view object that will act as the master.

For example, [Figure 5–24](#) shows the `CustomerID` attribute selected from the `PersonsVO` view object to perform this role. Click **Next**.

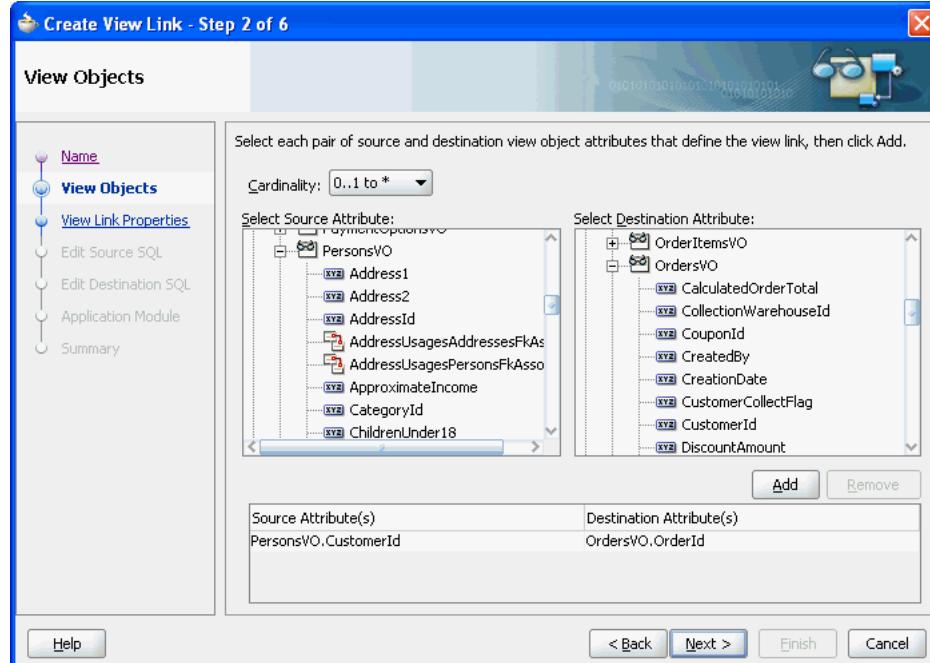
5. On the View Objects page, select a corresponding destination attribute from the view object that will act as the detail.

For example, if you want the detail query to show orders that were placed by the currently selected customer, select the `OrdersID` attribute in the `OrdersVO` to perform this role.

6. Click **Add** to add the matching attribute pair to the table of source and destination attribute pairs below. When you are finished defining master and detail link, click **Next**.

Figure 5–24 shows just one (CustomerId,OrderId) pair. However, if you require multiple attribute pairs to define the link between master and detail, repeat the steps for the View Objects page to add additional source-target attribute pairs.

Figure 5–24 Defining Source/Target Attribute Pairs While Creating a View Link

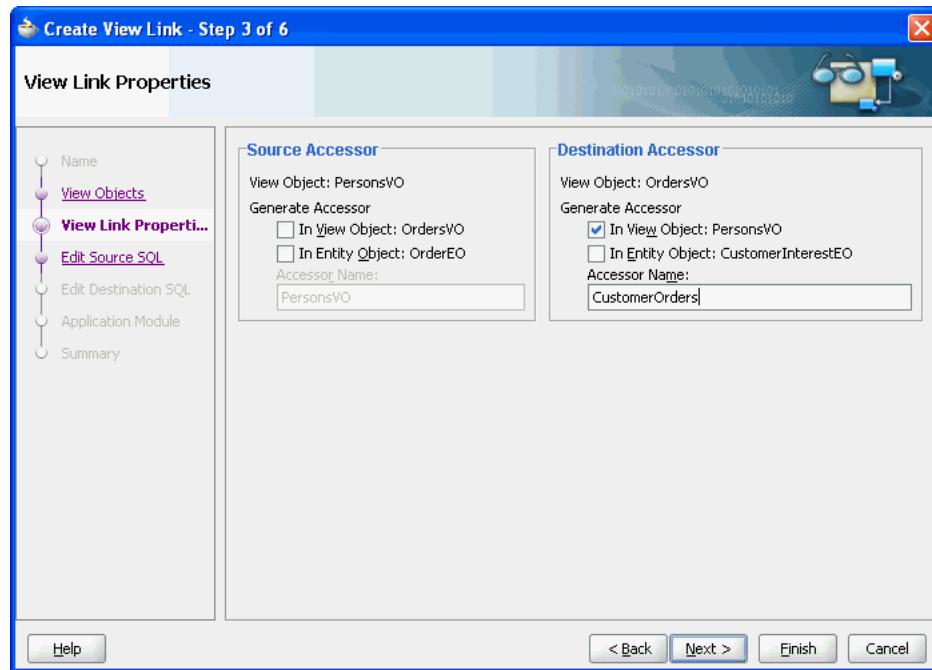


7. On the View Link Properties page, you can use the **Accessor Name** field to change the default name of the accessor that lets you programmatically access the destination view object.

By default, the accessor name will match the name of the destination view object. For example, you might change the default accessor name `OrdersVO1` to `CustomerOrders` to better describe the master-detail relationship that the accessor defines.

8. Also on the View Link Properties page, you control whether the view link represents a one-way relationship or a bidirectional one.

By default, a view link is a one-way relationship that allows the current row of the source (master) to access a set of related rows in the destination (detail) view object. For example, in Figure 5–25, the checkbox settings indicate that you'll be able to access a detail collection of rows from the `Orders` for the current row in the `Persons` view object, but not vice versa. In this case, this behavior is specified by the checkbox setting in the **Destination Accessor** group box for the `OrdersVO` view object (the **Generate Accessor In View Object: PersonsVO** box is selected) and checkbox setting in the **Source Accessor** group box for the `PersonsVO` view object (the **Generate Accessor In View Object: OrdersVO** box is not selected).

Figure 5–25 View Link Properties Control Name and Direction of Accessors

9. On the Edit Source SQL page, preview the view link SQL predicate that will be used at runtime to access the master row in the source view object and click **Next**.
10. On the Edit Destination SQL page, preview the view link SQL predicate that will be used at runtime to access the correlated detail rows from the destination view object for the current row in the source view object and click **Next**.
11. On the Application Module page, add the view link to the data model for the desired application module and click **Finish**.

By default the view link will not be added to the application module's data model. Later you can add the view link to the data model using the overview editor for the application module.

5.6.2 How to Create a Master-Detail Hierarchy for Entity-Based View Objects

Just as with read-only view objects, you can link entity-based view objects to other view objects to form master-detail hierarchies of any complexity. The only difference in the creation steps involves the case when both the master and detail view objects are entity-based view objects and their respective entity usages are related by an association. In this situation, since the association captures the set of source and destination attribute pairs that relate them, you create the view link just by indicating which association it should be based on.

To create an association-based view link, you use the Create View Link wizard.

To create an association-based view link

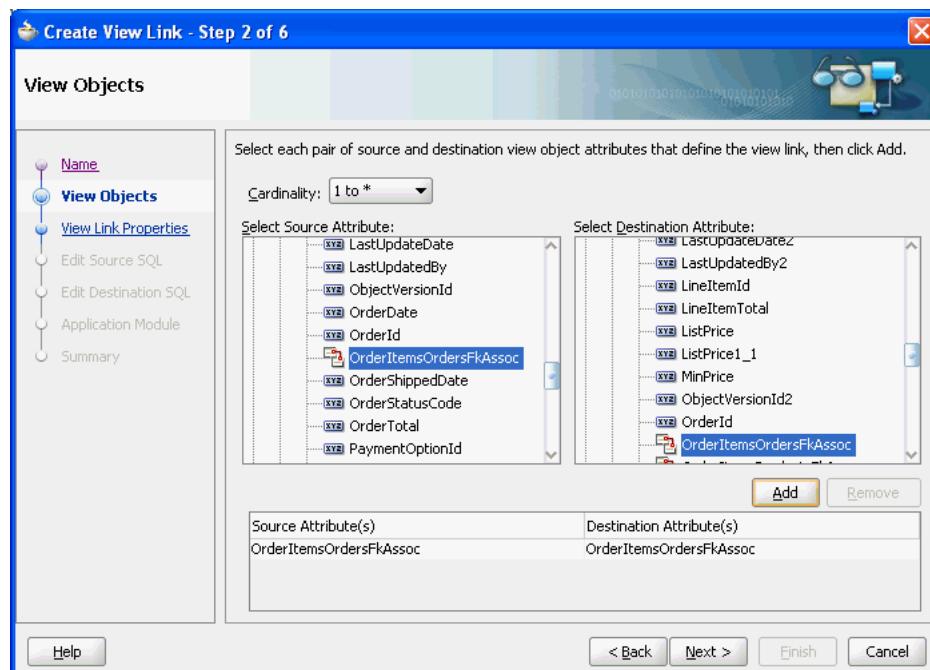
1. In the Application Navigator, right-click the project in which you want to create the view object and choose **New**.

To avoid having to type in the package name in the Create View Link wizard, you can choose **New View Link** on the context menu of the **links** package node in the Application Navigator.

2. In the New Gallery, expand **Business Tier**, select **ADF Business Components**, then **View Link**, and click **OK**.
3. In the Create View Link wizard, on the Name page, supply a package and a component name.
4. On the View Objects page, in the **Select Source Attribute** tree expand the source view object in the desired package. In the **Select Destination Attribute** tree expand the destination view object.
For entity-based view objects, notice that in addition to the view object attributes, relevant associations also appear in the list.
5. Select the same association in both **Source** and **Destination** trees. Then click **Add** to add the association to the table below.

For example, [Figure 5–26](#) shows the same `OrderItemsOrdersFkAssoc` association in both **Source** and **Destination** trees selected.

Figure 5–26 Master and Detail Related by an Association Selection



6. Click **Finish**.

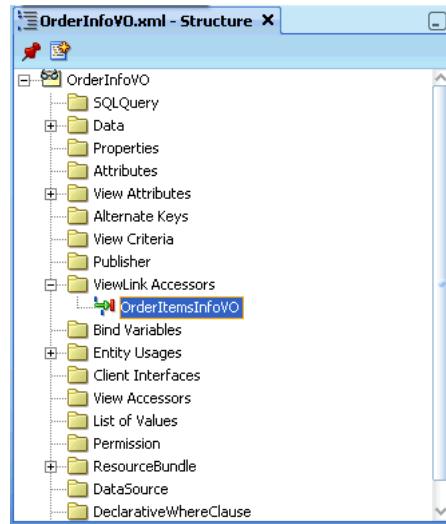
5.6.3 What Happens When You Create Master-Detail Hierarchies Using View Links

When you create a view link or an association-based view link, JDeveloper creates the XML component definition file that represents its declarative settings and saves it in the directory that corresponds to the name of its package. For example, if the view link is named `OrderInfoToOrderItemsInfo` and it appears in the `queries.links` package, then the XML file created will be

`./queries/link/OrderInfoToOrderItemsInfo.xml` under the project's source path. This XML file contains the declarative information about the source and target attribute pairs you've specified and, in the case of an association-based view link, contains the declarative information about the association that relates the source and target view objects you've specified.

In addition to saving the view link component definition itself, JDeveloper also updates the XML definition of the *source* view object in the view link relationship to add information about the view link accessor you've defined. As a confirmation of this, you can select the source view object in the Application Navigator and inspect its details in the Structure window. As shown in [Figure 5–27](#), you can see the defined accessor in the **ViewLink Accessors** node for the OrderItemsInfoVO source view object of the OrderInfoToOrderItemsInfo view link.

Figure 5–27 View Object with View Link Accessor in the Structure Window



Note: A view link defines a basic master-detail relationship between two view objects. However, by creating more view links you can achieve master-detail hierarchies of any complexity, including:

- Multilevel master-detail-detail
- Master with multiple (peer) details
- Detail with multiple masters

The steps to define these more complex hierarchies are the same as the ones covered in [Section 5.6.2, "How to Create a Master-Detail Hierarchy for Entity-Based View Objects"](#), you just need to create it one view link at time.

5.6.4 How to Enable Active Master-Detail Coordination in the Data Model

When you enable programmatic navigation to a row set of correlated details by defining a view link as described in [Section 5.6.2, "How to Create a Master-Detail Hierarchy for Entity-Based View Objects"](#), the view link plays a *passive* role, simply defining the information necessary to retrieve the coordinated detail row set when your code requests it. The view link accessor attribute is present and programmatically accessible in any result rows from any instance of the view link's source view object. In other words, programmatic access does not require modifying the application module's data model.

However, since master-detail user interfaces are such a frequent occurrence in enterprise applications, the view link can be also used in a more *active* fashion so you can avoid needing to coordinate master-detail screen programmatically. You opt to

have this active master-detail coordination performed by *explicitly* adding an instance of a view-linked view object to your application module's data model.

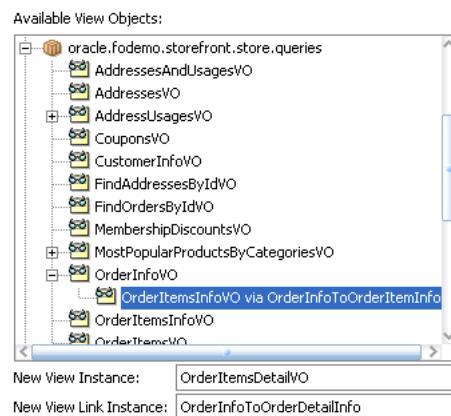
To enable active master-detail coordination, open the application module in the overview editor and select the Data Model page.

To add a detail instance of a view object:

1. In the Application Navigator, double-click the application module.
2. In the overview editor for the application module, open the **Data Model** page.
3. In the **Available View Objects** list, select the detail view object node that is indented beneath the master view object.

Note that the list shows the detail view object *twice*: once on its own, and once as detail view object via the view link. For example, in [Figure 5–28](#) you would select the detail view object `OrderItemsInfoVO` via `OrderInfoToOrderItemInfo` instead of the view object labeled as `OrderItemsInfoVO` (which, in this case, appears beneath the highlighted view object).

Figure 5–28 Detail View Object Selection from Available View Objects



4. Enter a name for the detail instance you're about to create in the **Name View Instance** field below the **Available View Objects** list.

For example, [Figure 5–28](#) shows the name `OrderItemsDetailVO` for the instance of the `OrderItemsInfoVO` view object that is a detail view.

5. In the **Data Model** list, select the instance of the view object that you want to be the actively-coordinating master.
6. Click **Add Instance** to add the detail instance to the currently selected master instance in the data model, with the name you've chosen.

For example, in [Figure 5–29](#), the **Data Model** list shows a master-detail hierarchy of view object instances with `OrderItemsDetailVO` as the detail view object.

Figure 5–29 Data Model with View Linked View Object

5.6.5 How to Test Master-Detail Coordination

To test active master-detail coordination, launch the Business Component Browser on the application module by choosing **Run** from its context menu in the Application Navigator. The Business Component Browser data model tree shows the view link instance that is actively coordinating the detail view object instance with the master view object instance. You can double-click the view link instance node in the tree to open a master-detail data view page in the Business Component Browser. Then, when you use the toolbar buttons to navigate in the master view object — changing the view object's current row as a result — the coordinated set of details is automatically refreshed and the user interface stays in sync.

If you double-click another view object that is not defined as a master and detail, a second tab will open to show its data; in that case, since it is not actively coordinated by a view link, its query is not constrained by the current row in the master view object.

For information about editing the data model and running the Business Component Browser, see [Section 6.3, "Testing View Object Instances Using the Business Component Browser"](#).

5.6.6 How to Access the Detail Collection Using the View Link Accessor

To work with view links effectively, you should also understand that view link accessor attributes return a RowSet object and that you can access a detail collection using the view link accessor programmatically.

5.6.6.1 Accessing Attributes of Row by Name

At runtime, the `getAttribute()` method on a Row object allows you to access the value of any attribute of that row in the view object's result set by name. The view link accessor behaves like an additional attribute in the current row of the source view object, so you can use the same `getAttribute()` method to retrieve its value. The only practical difference between a regular view attribute and a view link accessor attribute is its data type. Whereas a regular view attribute typically has a scalar data type with a value like 303 or ngreenbe, the value of a view link accessor attribute is a row set of zero or more correlated detail rows. Assuming that `curUser` is a Row object from some instance of the Orders view object, you can write a line of code to retrieve the detail row set of order items:

```
RowSet items = (RowSet) curUser.getAttribute("OrderItems");
```

Note: If you generate the custom Java class for your view row, the type of the view link accessor will be `RowIterator`. Since at runtime the return value will always be a `RowSet` object, it is safe to cast the view link attribute value to `RowSet`.

5.6.6.2 Programmatically Accessing a Detail Collection Using the View Link Accessor

Once you've retrieved the `RowSet` of detail rows using a view link accessor, you can loop over the rows it contains just as you would loop over a view object's row set of results, as shown in [Example 5–8](#).

Example 5–8 Programmatically Accessing a Detail Collection

```
while (items.hasNext()) {
    Row curItem = items.next();
    System.out.println("--> (" + curItem.getAttribute("LineItemId") + " ) " +
        curItem.getAttribute("LineItemTotal"));
}
```

For information about creating a test client, see [Section 6.4.6, "How to Access a Detail Collection Using the View Link Accessor"](#).

5.7 Working with View Objects in Declarative SQL Mode

At runtime, when ADF Business Components works with JDBC to pass a query to the database and retrieve the result, the mechanism to retrieve the data is the SQL query. As an alternative to creating view objects that specify a SQL statement at design time, you can create entity-based view objects that contain no SQL statements. This capability of the ADF Business Components design time and runtime is known as *declarative SQL mode*. When the data model developer works with the wizard or editor for a view object in declarative SQL mode, they require no knowledge of SQL. In declarative SQL mode, the view object's metadata causes the ADF Business Components runtime to generate the SQL query statements as follows:

- Generates SELECT and FROM lists based on the rendered web page's databound UI components' usage of one or more entity objects' attributes
- Optionally, generates a WHERE clause based on a view criteria that you add to the view object definition
- Optionally, generates an ORDERBY clause based on a sort criteria that you add to the view object definition.
- Optionally, augments the WHERE clause to support table joins based on named view criteria that you add to the view object definition
- Optionally, augments the WHERE clause to support master-detail view filtering based on a view criteria that you add to either the source or destination of a view link definition

Additionally, the SQL statement that a declarative SQL mode view object generates at runtime will be determined by the SQL flavor specified in the Business Component Project Properties dialog.

Note: Currently, the supported flavors for runtime SQL generation are SQL92 (ANSI) style and Oracle style. For information about setting the SQL flavor for your project, see [Section 3.3.1, "Choosing a Connection, SQL Flavor, and Type Map"](#).

Declarative SQL mode selection is supported in JDeveloper as a setting that you can apply either to the entire data model project or to individual view objects that you create. The ADF Business Components design time also allows you to override the declarative SQL mode project-level setting for any view object you create.

The alternatives to declarative SQL mode are normal mode and expert mode. When you work in either of those modes, the view object definitions you create at design time always contain the entire SQL statement based on the SQL flavor required by your application module's defined database connection. Thus the capability of SQL independence does not apply to view objects that you create in normal or expert mode. For information about using the wizard and editor to customize view objects when SQL is desired at design time, see [Section 5.2, "Populating View Object Rows from a Single Database Table"](#).

5.7.1 How to Create SQL-Independent View Objects with Declarative SQL Mode

All view objects that you create in JDeveloper rely on the same design time wizard and editor. However, when you enable declarative SQL mode, the wizard and editor change to support customizing the view object definition without requiring you to display or enter any SQL. For example, the Query page of the Create View Object wizard with declarative SQL mode enabled lacks the **Generated SQL** field present in normal mode.

Additionally, in declarative SQL mode, since the wizard and editor do not allow you to enter WHERE and ORDERBY clauses, you provide equivalent functionality by defining a view criteria and sort criteria respectively. In declarative SQL mode, these criteria appear in the view object metadata definition and will be converted at runtime to their corresponding SQL clause. When the databound UI component has no need to display filtered or sorted data, you may omit the view criteria or sort criteria from the view object definition.

Otherwise, after you enable declarative SQL mode, the basic procedure to create a view object with ensured SQL independence is the same as you would follow to create any entity-based view object. For example, you must still ensure that your view object encapsulates the desired entity object metadata to support the intended runtime query. As with any entity-based view object, the columns of the runtime-generated FROM list must relate to the attributes of one or more of the view object's underlying entity objects. In declarative SQL mode, you automatically fulfill this requirement when working with the wizard or editor when you add or remove the attributes of the entity objects on the view object definition.

Performance Tip: A view object instance configured to generate SQL statements dynamically will requery the database during page navigation if a subset of all attributes with the same list of key entity objects is used in the subsequent page navigation. Thus performance can be improved by activating a superset of all the required attributes to eliminate a subsequent query execution.

Thus there are no unique requirements for creating entity-based view objects in declarative SQL mode, nor does declarative SQL mode sacrifice any of the runtime

functionality of the normal mode counterpart. You can enable declarative SQL mode as a global preference so that it is the Create View Object wizard's default mode, or you can leave the setting disabled and select the desired mode directly in the wizard. The editor for a view object also lets you select and change the mode for an existing view object definition.

To enable declarative SQL mode for all new view objects:

1. From the main menu, choose **Tools > Preferences**.
2. In the Preferences dialog, expand the **Business Components** node and choose **View Objects**.
3. On the Business Components: View Object page, select **Enable declarative SQL mode for new objects** and click **OK**.

To predetermine how the FROM list will be generated at runtime you can select **Include all attributes in runtime-generated query**, as described in [Section 5.7.4, "How to Force Attribute Queries for Declarative SQL Mode View Objects"](#).

To create an entity-based view object in declarative SQL mode, use the Create View Object wizard, which is available from the New Gallery.

To create declarative SQL-based view objects:

1. In the Application Navigator, right-click the project in which you want to create the view objects and choose **New**.
2. In the New Gallery, expand **Business Tier**, select **ADF Business Components**, then **View Object**, and click **OK**.
3. On the Name page, enter a package name and a view object name. Keep the default setting **Updatable access through entity objects** enabled to indicate that you want this view object to manage data with its base entity object. Click **Next**. Any other choice for the data selection will disable declarative SQL mode in the Create View Object wizard.
4. On the Entity Objects page, select the entity object whose data you want to use in the view object. Click **Next**.

When you want to create a view object that joins entity objects, you can add secondary entity objects to the list. To create more complex entity-based view objects, see [Section 5.5.1, "How to Create Joins for Entity-Based View Objects"](#).

5. On the Attributes page, select at least one attribute from the entity usage in the **Available** list and shuttle it to the **Selected** list. Attributes you do not select will not be eligible for use in view criteria and sort criteria. Click **Next**.

You should select any attribute that you wish to customize in the Attribute Settings page or select for use in a view criteria or sort criteria in the Query page. Additionally, the tables that appear in the FROM list of the runtime-generated query will be limited to the tables corresponding to the attributes of the entity objects you select.

6. On the Attribute Settings page, optionally, use the **Select Attribute** dropdown list to switch between the view object attributes in order to change their names or any of their initial settings. Click **Next**.
7. On the Query page, select **Declarative** in the **Query Mode** dropdown list if it is not already displayed. The wizard changes to declarative SQL mode.

If you did not select **Enable declarative SQL mode for new objects**, in the Preferences dialog, the wizard displays the default query mode, **Normal**.

Changing the mode to **Declarative** in the wizard allows you to override the default mode for this single view object.

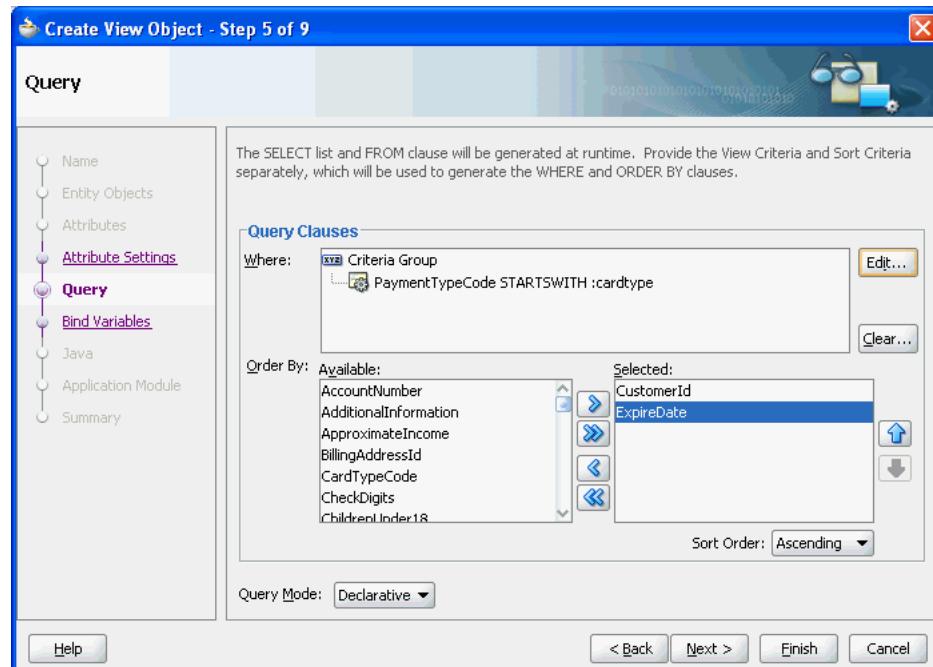
8. Optionally, define **Where** and **Order By** criteria to filter and order the data as required. At runtime, ADF Business Components automatically generates the corresponding SQL statements based on the criteria you create.

Click **Edit** next to the **Where** field to define the view criteria you will use to filter the data. The view criteria you enter will be converted at runtime to a WHERE clause that will be enforced on the query statement. For information about specifying view criteria, see [Section 5.10, "Working with Named View Criteria"](#).

In the **Order By** field select the desired attribute in the **Available** list and shuttle it to the **Selected** list. Attributes you do not select will not appear in the SQL ORDERBY clause generated at runtime. Add additional attributes to the **Selected** list when you want the results to be sorted by more than one column. Arrange the selected attributes in the list according to their sort precedence. Then for each sort attribute, assign whether the sort should be performed in ascending or descending order. Assigning the sort order to each attribute ensures that attributes ignored by the UI component still follow the intended sort order.

For example, as shown in [Figure 5–30](#), to limit the CustomerCardStatus view object to display only the rows in the CUSTOMERS table for customers with a specific credit card code, the view criteria in the **Where** field limits the CardTypeCode attribute to a runtime-determined value. To order the data by customer ID and the customer's card expiration date, the **Order By** field identifies those attributes in the **Selected** list.

Figure 5–30 Creating View Object Wizard, Query Page with Declarative Mode Selected



9. Click **Finish**.

5.7.2 How to Filter Declarative SQL-Based View Objects When Table Joins Apply

When you create an entity-based view object you can reference more than one entity object in the view object definition. In the case of view objects you create in declarative SQL mode, whether the base entity objects are activated from the view object definition will depend on the requirements of the databound UI component at runtime. If the UI component displays attribute values from multiple entity objects, then the SQL generated at runtime will contain a `JOIN` operation to query the appropriate tables.

Just as with any view object that you create, it is possible to filter the results from table joins by applying named view criteria. In the case of normal mode view objects, all entity objects and their attributes will be referenced by the view object definition and therefore will be automatically included in the view object's SQL statement. However, by delaying the SQL generation until runtime with declarative SQL mode, there is no way to know whether the view criteria should be applied.

Note: In declarative SQL mode, you can define a view criteria to specify the `WHERE` clause (optional) when you create the view object definition. This type of view criteria when it exists will always be applied at runtime. For a description of this usage of the view criteria, see [Section 5.7.1, "How to Create SQL-Independent View Objects with Declarative SQL Mode"](#).

Because a SQL `JOIN` may not always result from a view object defined in declarative SQL mode with multiple entity objects, named view criteria that you define to filter query results should be applied conditionally at runtime. In other words, named view criteria that you create for declarative SQL-based view objects need not be applied as required, automatic filters. To support declarative SQL mode, named view criteria that you apply to a view object created in declarative SQL mode can be set to apply only on the condition that the UI component is bound to the attributes referenced by the view criteria. The named view criteria once applied will, however, support the UI component's need to display a filtered result set.

You use the Edit View Criteria dialog to create the named view criteria and enable its conditional usage by setting the `appliedIfJoinSatisfied` property in the Property Inspector.

To define a view criteria to filter only when the join is satisfied:

1. Create the view object with declarative SQL mode enabled as described in [Section 5.7.1, "How to Create SQL-Independent View Objects with Declarative SQL Mode"](#).
2. In the Application Navigator, double-click the view object.
3. In the overview editor for the view object, select **Query** from the navigation list and expand **View Criteria**.
4. In the **View Criteria** section, click the **Create New View Criteria** icon to open the Edit View Criteria dialog.
5. Create the view criteria as described in [Section 5.10.1, "How to Create Named View Criteria Declaratively"](#).
6. After creating the view criteria, select it in the **View Criteria** section of **Query** page of the overview editor.

7. With the view criteria selected, open the Property Inspector and set the **appliedIfJoinSatisfied** property to **true**.

The property value **true** means you want the view criteria to be applied only on the condition that the UI component requires the attributes referenced by the view criteria. The default value **false** means that the view criteria will automatically be applied at runtime. In the case of declarative SQL mode-based view objects, the value **true** ensures that the query filter will be appropriate to needs of the view object's databound UI component.

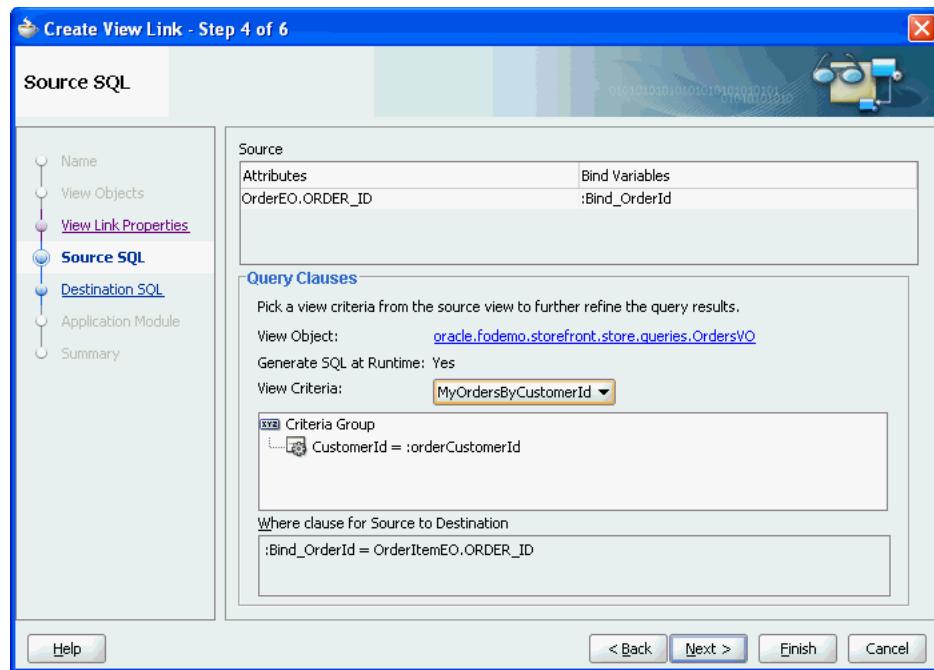
5.7.3 How to Filter Master-Detail Related View Objects with Declarative SQL Mode

Just as with normal mode view objects, you can link view objects that you create in declarative SQL mode to other view objects to form master-detail hierarchies of any complexity. The steps to create the view links are the same as with any other entity-based view object, as described in [Section 5.6.2, "How to Create a Master-Detail Hierarchy for Entity-Based View Objects"](#). However, in the case of view objects that you create in declarative SQL mode, you can further refine the view object results in the Source SQL or Destination SQL dialog for the view link by selecting a previously defined view criteria in the Create View Link wizard or the overview editor for the view link.

To define a view criteria for view link source or view link destination:

1. Create the view objects in declarative SQL mode as described in [Section 5.7.1, "How to Create SQL-Independent View Objects with Declarative SQL Mode"](#).
2. In the overview editor for the view objects, define the desired view criteria for either the source (master) view object or the destination (detail) view object as described in [Section 5.7.2, "How to Filter Declarative SQL-Based View Objects When Table Joins Apply"](#).
3. Create the view link as described in [Section 5.6.2, "How to Create a Master-Detail Hierarchy for Entity-Based View Objects"](#) and perform one of these additional steps:
 - On the SQL Source page, select a previously defined view criteria to filter the master view object. Click **Next**.
 - On the Destination SQL page, select a previously defined view criteria to filter the detail view object.

[Figure 5–30](#) shows a view criteria that filters the master view object based on customer IDs.

Figure 5–31 Filtering a View Link in Declarative SQL Mode

4. After you create the view link, you can also select a previously defined view criteria. In the overview editor navigation list, select **Query** and expand the **Source** or **Destination** sections. In the **View Criteria** dropdown list, select the desired view criteria. The dropdown list will be empty if no view criteria exist for the view object.

If the overview editor does not display a dropdown list for view criteria selection, then the view objects you selected for the view link were not created in declarative SQL mode. For view objects created in normal or expert mode, you must edit the WHERE clause to filter the data as required.

5.7.4 How to Force Attribute Queries for Declarative SQL Mode View Objects

Typically, when you define a declarative SQL mode view object, the attributes that get queried at runtime will be determined by the requirements of the databound UI component as it is rendered in the web page. This is the runtime-generation capability that makes view objects independent of the design time database's SQL flavor. However, you may also need to execute the view object programmatically without exposing it to an ADF data binding in the UI. In this case, you can enable the **Include all attributes in runtime-generated query** option to ensure that a programmatically executed view object has access to all of the entity attributes.

Note: Be careful to limit the use of the **Include all attributes in runtime-generated query** option to programmatically executed view objects. If you expose the view object with this setting enabled to a databound UI component, the runtime query will include all attributes.

The **Include all attributes in runtime-generated query** option can be specified as a global preference setting or as a setting on individual view objects. Both settings may be used in these combinations:

- Enable the global preference so that every view object you create includes all attributes in the runtime query statement.
- Enable the global preference, but disable the setting on view objects that will not be executed programmatically and therefore should not include all attributes in the runtime query statement.
- Disable the global preference (default), but enable the setting on view objects that will be executed programmatically and therefore should include all attributes in the runtime query statement.

To set the global preference to include all attributes in the query:

1. From the main menu, choose **Tools > Preferences**.
2. In the Preferences dialog, expand **Business Components** and select **View Objects**.
3. On the Business Components: View Object page, select **Enable declarative SQL for new objects**.
4. Select **Include all attributes in runtime-generated query** to force all attributes of the view object's underlying entity objects to participate in the query and click **OK**.

Enabling this option sets a flag in the view object definition but you will still need to add entity object selections and entity object attribute selections to the view object definition.

You can change the view object setting in the **Tuning** section of the overview editor's General page. The overview editor only displays the **Include all attributes in runtime-generated query** option if you have created the view object in declarative SQL mode.

To set the view object-specific preference to include all attributes in the query:

1. When you want to force all attributes for specific view objects, create the view object in the Create View Object wizard and be sure that you have enabled declarative SQL mode.
- You can verify this in the overview editor. In the overview editor navigation list, select **Query** and click the **Edit SQL Query** icon along the top of the page. Verify that the **SQL Mode** dropdown list shows the selection **Declarative SQL Mode**.
2. In the overview editor navigation list, select **General** and expand the **Tuning** section.
 3. Select **Include all attributes in runtime-generated query** to force all attributes of the view object's underlying entity objects to participate in the query and click **OK**.

Enabling this option sets a flag in the view object definition but you will still need to add entity object selections and entity object attribute selections to the view object definition.

5.7.5 What Happens When You Create a View Object in Declarative SQL Mode

When you create the view object in declarative SQL mode, three properties get added to the view object's metadata: `SelectListFlags`, `FromListFlags`, and `WhereFlags`. Properties that are absent in declarative SQL mode are the normal mode view object's `SelectList`, `FromList`, and `Where` properties, which contain the actual SQL statement (or, for expert mode, the `SQLQuery` element). [Example 5–9](#) shows the three view object metadata flags that get enabled in declarative SQL mode

to ensure that SQL will be generated at runtime instead of specified as metadata in the view object's definition.

Example 5–9 View Object Metadata with Declarative SQL Mode Enabled

```
<ViewObject
    xmlns="http://xmlns.oracle.com/bc4j"
    Name="CustomerCardStatus"
    SelectListFlags="1"
    FromListFlags="1"
    WhereFlags="1"
    ...
    ...
```

Similar to view objects that you create in either normal or expert mode, the view object metadata also includes a `ViewAttribute` element for each attribute that you select in the Attribute page of the Create View Object wizard. However, in declarative SQL mode, when you "select" attributes in the wizard (or add an attribute in the overview editor), you are not creating a `FROM` or `SELECT` list in the design time. The attribute definitions that appear in the view object metadata only determine the list of potential entities and attributes that will appear in the runtime-generated statements. For information about how ADF Business Components generates these SQL lists, see [Section 5.7.6, "What Happens at Runtime"](#).

[Example 5–10](#) shows the additional features of declarative SQL mode view objects, including the optional declarative WHERE clause (`DeclarativeWhereClause` element) and the optional declarative ORDERBY clause (`SortCriteria` element).

Example 5–10 View Object Metadata: Declarative View Criteria and Sort Criteria

```
<DeclarativeWhereClause>
    <ViewCriteria
        Name="CustomerStatusWhereCriteria"
        ViewObjectName="oracle.fodemo.storefront.store.queries.CustomerCardStatus"
        Conjunction="AND"
        Mode="3"
        AppliedIfJoinSatisfied="false">
        <ViewCriteriaRow
            Name="vcrow60">
            <ViewCriteriaItem
                Name="CardTypeCode"
                ViewAttribute="CardTypeCode"
                Operator="STARTSWITH"
                Conjunction="AND"
                Required="Optional">
                <ViewCriteriaItemValue
                    Value=":cardtype"
                    IsBindVarValue="true"/>
            </ViewCriteriaItem>
        </ViewCriteriaRow>
    </ViewCriteria>
</DeclarativeWhereClause>
<SortCriteria>
    <Sort
        Attribute="CustomerId" />
<Sort
        Attribute="CardTypeCode" />
</SortCriteria>
```

5.7.6 What Happens at Runtime

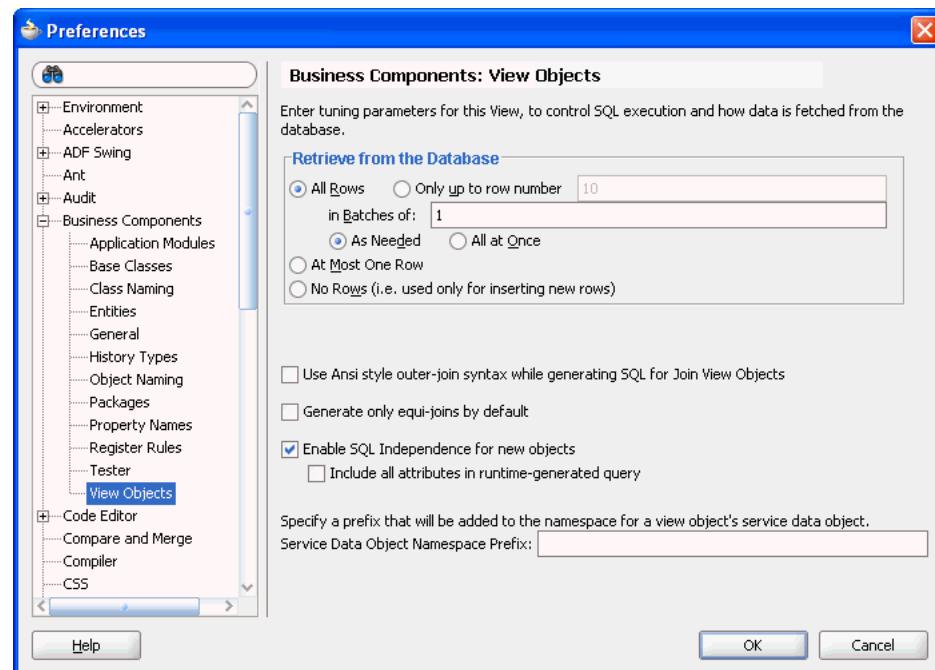
At runtime, when a declarative SQL mode query is generated, ADF Business Components determines which attributes were defined from the metadata ViewCriteria element and SortCriteria element. It then uses these attributes to generate the WHERE and ORDERBY clauses. Next, the runtime generates the FROM list based on the tables corresponding to the entity usages defined by the metadata ViewAttribute elements. Finally, the runtime builds the SELECT statement based on the attribute selection choices the end user makes in the UI. As a result, the view object in declarative SQL mode generates all SQL clauses entirely at runtime. The runtime-generated SQL statements will be based on the flavor that appears in the project properties setting. Currently, the runtime supports SQL92 (ANSI) style and Oracle style flavors.

5.7.7 What You May Need to Know About Overriding Declarative SQL Mode Defaults

JDeveloper lets you control declarative SQL mode for all new view objects you add to your data model project or for individual view objects you create or edit. These settings may be used in these combinations:

- Enable the global preference in the Preferences dialog (select Tools > Preferences). Every view object you create will delay SQL generation until runtime. [Figure 5–32](#) shows the global preference **Enable declarative SQL for new objects** set to enabled.
- Enable the global preference in the Preferences dialog, but change the SQL mode for individual view objects. In this case, unless you change the SQL mode, the view objects you create will delay SQL generation until runtime.
- Disable the global preference (default) in the Preferences dialog, but select declarative SQL mode for individual view objects. In this case, unless you change the SQL mode, view objects you create will contain SQL statements.

Figure 5–32 Preferences Dialog with Declarative SQL Mode Enabled



To edit the SQL mode for a view object you have already created, open the Query page in the Edit Query dialog and select **Declarative** from the **SQL Mode** dropdown list. To display the Edit Query dialog, open the view object in the overview editor, select **Query** from the navigation list and click the **Edit SQL Query** icon. The same option appears in the Query page of the Create View Object wizard.

5.7.8 What You May Need to Know About Working Programmatically with Declarative SQL Mode View Objects

As a convenience to developers, the view object implementation API allows individual attributes to be selected and deselected programmatically. This API may be useful in combination with the view objects you create in declarative SQL mode and intend to execute programmatically. [Example 5-11](#) shows how to call `selectAttributeDefs()` on the view object when you want to add a subset of attributes to those already configured with SQL mode enabled.

Example 5-11 ViewObjectImpl API with SQL Mode View Objects

```
ApplicationModule am = Configuration.createRootApplicationModule(amDef, config);
ViewObjectImpl vo = (ViewObjectImpl) am.findViewObject("CustomerVO");
vo.resetSelectedAttributeDefs(false);
vo.selectAttributeDefs(new String[] {"FirstName", "LastName"});
vo.executeQuery();
```

The call to `selectAttributeDefs()` adds the attributes in the array to a private member variable of `ViewObjectImpl`. A call to `executeQuery()` transfers the attributes in the private member variable to the actual select list. It is important to understand that these `ViewObjectImpl` attribute calls are not applicable to the client layer and are only accessible inside the `Impl` class of the view object on the middle tier.

Additionally, you might call `unselectAttributeDefs()` on the view object when you want to deselect a small subset of attributes after *enabling* the **Include all attributes in runtime-generated query** option. Alternatively, you can call `selectAttributeDefs()` on the view object to select a small subset of attributes after *disabling* the **Include all attributes in runtime-generated query** option.

Caution: Be careful not to expose a declarative SQL mode view object executed with this API to the UI since only the value of the **Include all attributes in runtime-generated query** option will be honored.

5.8 Working with View Objects in Expert Mode

When defining entity-based view objects, you can fully specify the WHERE and ORDER BY clauses, whereas, by default, the FROM clause and SELECT list are automatically derived. The names of the tables related to the participating entity usages determine the FROM clause, while the SELECT list is based on the:

- Underlying column names of participating entity-mapped attributes
- SQL expressions of SQL-calculated attributes

When you require full control over the SELECT or FROM clause in a query, you can enable expert mode.

5.8.1 How to Customize SQL Statements in Expert Mode

To enable expert mode, select **Expert Mode** from the **SQL Mode** dropdown list on the SQL Statement page of the Create View Object wizard or Edit View Object dialog.

5.8.2 How to Name Attributes in Expert Mode

If your SQL query includes a calculated expression, use a SQL alias to assist the Create View Object wizard in naming the column with a Java-friendly name. [Example 5–12](#) shows a SQL query that includes a calculated expression.

Example 5–12 SQL Query with Calculated Expression

```
select PERSON_ID, EMAIL,
       SUBSTR(FIRST_NAME,1,1) || '.' || LAST_NAME
  from PERSONS
 order by EMAIL
```

[Example 5–13](#) uses a SQL alias to assist the Create View Object wizard in naming the column with a Java-friendly name.

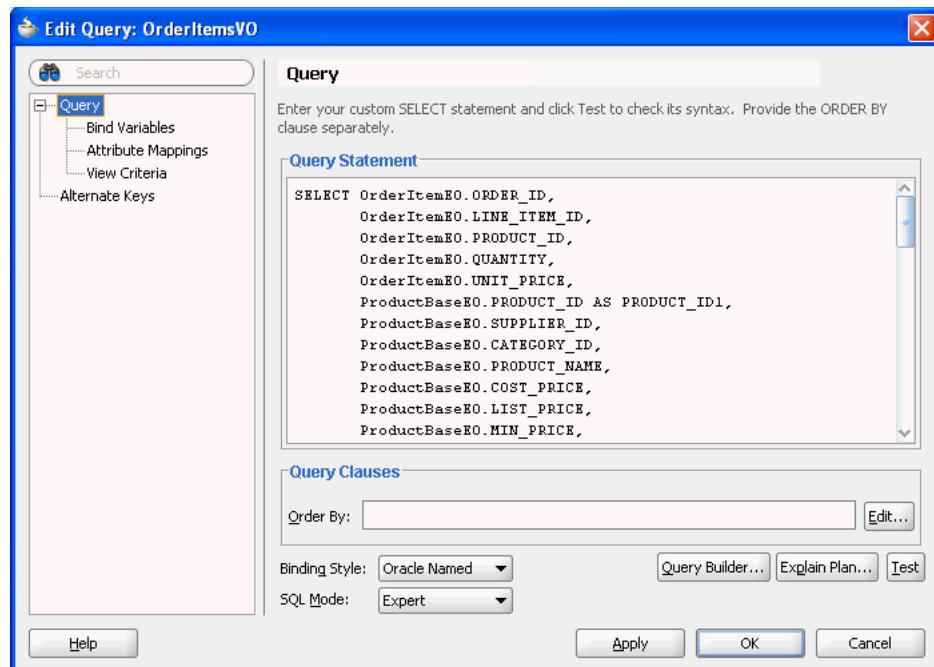
Example 5–13 SQL Query with SQL Alias

```
select PERSON_ID, EMAIL,
       SUBSTR(FIRST_NAME,1,1) || '.' || LAST_NAME AS USER_SHORT_NAME
  from PERSONS
 order by EMAIL
```

5.8.3 What Happens When You Enable Expert Mode

When you enable expert mode, the read-only **Generated Statement** section of the Query page becomes a fully editable **Query Statement** text box, displaying the full SQL statement. Using this text box, you can change every aspect of the SQL query.

For example, [Figure 5–33](#) shows the Query page of the Edit View Object dialog for the OrderItems view object. It's an expert mode, entity-based view object that references a PL/SQL function decode that obtains its input values from an expression set on the ShippingCost attribute.

Figure 5–33 OrderItems Expert Mode View Object

5.8.4 What You May Need to Know About Expert Mode

When you define a SQL query using expert mode in the Edit Query dialog, you type a SQL language statement directly into the editor. Using this mode places some responsibility on the Business Components developer to understand how the view object handles the metadata resulting from the query definition. Review the following information to familiarize yourself with the behavior of the Edit Query dialog that you use in expert mode.

5.8.4.1 Expert Mode Provides Limited Attribute Mapping Assistance

The automatic cooperation of a view object with its underlying entity objects depends on correct attribute-mapping metadata saved in the XML component definition. This information relates the view object attributes to corresponding attributes from participating entity usages. JDeveloper maintains this attribute mapping information in a fully automatic way for normal entity-based view objects. However, when you decide to use expert mode with a view object, you need to pay attention to the changes you make to the SELECT list. That is the part of the SQL query that directly relates to the attribute mapping. Even in expert mode, JDeveloper continues to offer some assistance in maintaining the attribute mapping metadata when you do the following to the SELECT list:

- Reorder an expression without changing its column alias
JDeveloper reorders the corresponding view object attribute and maintains the attribute mapping.
- Add a new expression
- JDeveloper adds a new SQL-calculated view object attribute with a corresponding camel-capped name based on the column alias of the new expression.
- Remove an expression

JDeveloper converts the corresponding SQL-calculated or entity-mapped attribute related to that expression to a transient attribute.

However, if you rename a column alias in the SELECT list, JDeveloper has no way to detect this, so it is treated as if you removed the old column expression and added a new one of a different name.

After making any changes to the SELECT list of the query, visit the Attribute Mappings page to ensure that the attribute-mapping metadata is correct. The table on this page, which is disabled for view objects in normal mode, becomes enabled for expert mode view objects. For each view object attribute, you will see its corresponding SQL column alias in the table. By clicking into a cell in the **View Attributes** column, you can use the dropdown list that appears to select the appropriate entity object attribute to which any entity-mapped view attributes should correspond.

Note: If the view attribute is SQL-calculated or transient, a corresponding attribute with a "SQL" icon appears in the **View Attributes** column to represent it. Since neither of these type of attributes are related to underlying entity objects, there is no entity attribute related information required for them.

5.8.4.2 Expert Mode Drops Custom Edits

When you disable expert mode for a view object, it will return to having its SELECT and FROM clause be derived again. JDeveloper warns you that doing this might cause your custom edits to the SQL statement to be lost. If this is what you want, after acknowledging the alert, your view object's SQL query reverts back to the default.

5.8.4.3 Expert Mode Ignores Changes to SQL Expressions

Consider a Products view object with a SQL-calculated attribute named Shortens whose SQL expression you defined as SUBSTR (NAME , 1 , 10) . If you switch this view object to expert mode, the **Query Statement** box will show a SQL query similar to the one shown in [Example 5–14](#).

Example 5–14 SQL-Calculated Attribute Expression in Expert Mode

```
SELECT Products.PROD_ID,
       Products.NAME,
       Products.IMAGE,
       Products.DESCRIPTION,
       SUBSTR(NAME,1,10) AS SHORT_NAME
  FROM PRODUCTS Products
```

If you go back to the attribute definition for the Shortens attribute and change the **SQL Expression** field from SUBSTR (NAME , 1 , 10) to SUBSTR (NAME , 1 , 15) , then the change will be saved in the view object's XML component definition. Note, however, that the SQL query in the **Query Statement** box will remain as the original expression. This occurs because JDeveloper never tries to *modify* the text of an expert mode query. In expert mode, the developer is in full control. JDeveloper attempts to adjust metadata as a result of some kinds of changes you make yourself to the expert mode SQL statement, but it does not perform the reverse. Therefore, if you change view object metadata, the expert mode SQL statement is not updated to reflect it.

Therefore, you need to update the expression in the expert mode SQL statement itself. To be completely thorough, you should make the change *both* in the attribute metadata *and* in the expert mode SQL statement. This would ensure — if you (or another

developer on your team) ever decides to toggle expert mode *off* at a later point in time — that the automatically derived SELECT list would contain the correct SQL-derived expression.

Note: If you find you had to make numerous changes to the view object metadata of an expert mode view object, you can avoid having to manually translate any effects to the SQL statement by copying the text of your customized query to a temporary backup file. Then, you can disable expert mode for the view object and acknowledge the warning that you will lose your changes. At this point JDeveloper will rederive the correct generated SQL statement based on all the new metadata changes you've made. Finally, you can enable expert mode once again and reapply your SQL customizations.

5.8.4.4 Expert Mode Returns Error for SQL Calculations that Change Entity Attributes

When changing the SELECT list expression that corresponds to *entity-mapped* attributes, don't introduce SQL calculations into SQL statements that change the value of the attribute when retrieving the data. To illustrate the problem that will occur if you do this, consider the query for a simple entity-based view object named Products shown in [Example 5–15](#).

Example 5–15 Query Statement Without SQL-Calculated Expression

```
SELECT Products.PROD_ID,
       Products.NAME,
       Products.IMAGE,
       Products.DESCRIPTION
  FROM PRODUCTS Products
```

Imagine that you wanted to limit the name column to display only the first ten characters of the name of a product query. The correct way to do that would be to introduce a new SQL-calculated field, such as ShortName with an expression like SUBSTR(Products.NAME, 1, 10). One way you should *avoid* doing this is to switch the view object to expert mode and change the SELECT list expression for the entity-mapped NAME column to the include the SQL-calculated expression, as shown in [Example 5–16](#).

Example 5–16 Query Statement With SQL-Calculated Expression

```
SELECT Products.PROD_ID,
       SUBSTR(Products.NAME, 1, 10) AS NAME,
       Products.IMAGE,
       Products.DESCRIPTION
  FROM PRODUCTS Products
```

This alternative strategy would initially appear to work. At runtime, you see the truncated value of the name as you are expecting. However, if you modify the row, when the underlying entity object attempts to lock the row it does the following:

- Issues a SELECT FOR UPDATE statement, retrieving all columns as it tries to lock the row.
- If the entity object successfully locks the row, it compares the original values of all the persistent attributes in the entity cache as they were last retrieved from the

database with the values of those attributes just retrieved from the database during the lock operation.

- If any of the values differs, then the following error is thrown:

```
(oracle.jbo.RowInconsistentException)
JBO-25014: Another user has changed the row with primary key [...]
```

If you see an error like this at runtime even though you are the *only* user testing the system, it is most likely due to your inadvertently introducing a SQL function in your expert mode view object that changed the selected value of an entity-mapped attribute. In [Example 5–16](#), the SUBSTR(Products.NAME, 1, 10) function introduced causes the original selected value of the Name attribute to be truncated. When the row-lock SQL statement selects the value of the NAME column, it will select the entire value. This will cause the comparison shown in [Example 5–16](#) to fail, producing the "phantom" error that another user has changed the row.

The same thing would happen with NUMBER-valued or DATE-valued attributes if you inadvertently apply SQL functions in expert mode to truncate or alter their retrieved values for entity-mapped attributes.

Therefore, if you need to present altered versions of entity-mapped attribute data, introduce a new SQL-calculated attribute with the appropriate expression to handle the task.

5.8.4.5 Expert Mode Retains Formatting of SQL Statement

When you change a view object to expert mode, its XML component definition changes from storing parts of the query in separate XML attributes, to saving the entire query in a single <SQLQuery> element. The query is wrapped in an XML CDATA section to preserve the line formatting you may have done to make a complex query be easier to understand.

5.8.4.6 Expert Mode Wraps Queries as Inline Views

If your expert-mode view object:

- Contains a ORDERBY clause specified in the **Order By** field of the Query Clauses page at design time, or
- Has a dynamic WHERE clause or ORDERBY clause applied at runtime using `setWhereClause()` or `setOrderByClause()`

then its query gets nested into an inline view before applying these clauses. For example, suppose your expert mode query was defined like the one shown in [Example 5–17](#).

Example 5–17 Expert Mode Query Specified At Design Time

```
select PERSON_ID, EMAIL, FIRST_NAME, LAST_NAME
from PERSONS
union all
select PERSON_ID, EMAIL, FIRST_NAME, LAST_NAME
from INACTIVE_PERSONS
```

Then, at runtime, when you set an additional WHERE clause like `email = :TheUserEmail`, the view object nests its original query into an inline view like the one shown in [Example 5–18](#).

Example 5–18 Runtime-Generated Query With Inline Nested Query

```
SELECT * FROM(
select PERSON_ID, EMAIL, FIRST_NAME, LAST_NAME
from PERSONS
union all
select PERSON_ID, EMAIL, FIRST_NAME, LAST_NAME
from INACTIVE_PERSONS) QRSLT
```

And, the view object adds the dynamic WHERE clause predicate at the end, so that the final query the database sees looks like the one shown in [Example 5–19](#).

Example 5–19 Runtime-Generated Query With Dynamic WHERE Clause

```
SELECT * FROM(
select PERSON_ID, EMAIL, FIRST_NAME, LAST_NAME
from PERSONS
union all
select PERSON_ID, EMAIL, FIRST_NAME, LAST_NAME
from INACTIVE_PERSONS) QRSLT
WHERE email = :TheUserEmail
```

This query "wrapping" is necessary in general for expert mode queries, because the original query could be arbitrarily complex, including SQL UNION, INTERSECT, MINUS, or other operators that combine multiple queries into a single result. In those cases, simply "gluing" the additional runtime WHERE clause onto the end of the query text could produce unexpected results. For example, the clause might apply only to the *last* of several UNION'ed statements. By nesting the original query verbatim into an inline view, the view object guarantees that your additional WHERE clause is correctly used to filter the results of the original query, regardless of how complex it is.

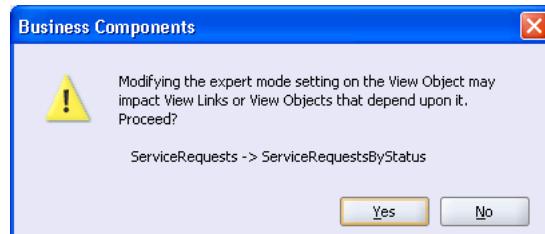
5.8.4.7 Limitation of Inline View Wrapping at Runtime

Inline view wrapping of expert mode view objects, limits a dynamically added WHERE clause to refer only to columns in the SELECT list of the original query. To avoid this limitation, when necessary you can disable the use of the inline view wrapping by calling `setNestedSelectForFullSql(false)`.

5.8.4.8 Expert Mode Changes May Affect Dependent Objects

When you modify a view object query to be in expert mode after you have already created the view links that involve that view object or after you created other view objects that extend the view object, JDeveloper will warn you with the alert shown in [Figure 5–34](#). The alert reminds you that you should revisit these dependent components to ensure their SQL statements still reflect the correct query.

Figure 5–34 Proactive Reminder to Revisit Dependent Components



For example, if you were to modify the `ServiceRequests` view object to use expert mode, because the `ServiceRequestsByStatus` view object extends it, you need to

revisit the extended component to ensure that its query still logically reflects an extension of the modified parent component.

5.9 Working with Bind Variables

Bind variables provide you with the means to supply attribute values at runtime to the view object or view criteria. All bind variables are defined at the level of the view object and used in one of the following ways:

- You can type the bind variable directly into the WHERE clause of your view object's query to include values that might change from execution to execution. In this case, bind variables serve as placeholders in the SQL string whose value you can easily change at runtime without altering the text of the SQL string itself. Since the query doesn't change, the database can efficiently reuse the same parsed representation of the query across multiple executions, which leads to higher runtime performance of your application.
- You can select the bind variable from a selection list to define the attribute value for a view criteria in the Edit View Criteria dialog you open on the view object. In this case, the bind variables allow you to change the values for attributes you will use to filter the view object row set. For more information about filtering view object row sets, see [Section 5.10, "Working with Named View Criteria"](#).

If the view criteria is to be used in a seeded search, you have the option of making the bind variable updatable by the end user. With this updatable option, end users will be expected to enter the value in a search form corresponding to the view object query.

Bind variables that you add to a WHERE clause require a valid value at runtime, or a runtime exception error will be thrown. In contrast, view criteria execution need not require the bind variable value if the view criteria item for which the bind variable is assigned is not required. To enforce this desired behavior, the Bind Variable dialog lets you can specify whether a bind variable is required or not.

You can define a default value for the bind variable or write scripting expressions for the bind variable that includes dot notation access to attribute property values. Expressions are based on the Groovy scripting language, as described in [Section 3.6, "Overview of Groovy Support"](#).

5.9.1 How to Add Bind Variables to a View Object Definition

To add a named bind variable to a view object, use the query page of the overview editor for the view object. You can define as many bind variables as you need.

To define a named bind variable:

1. In the Application Navigator, double-click the view object to open the overview editor.
2. In the overview editor navigation list, select **Query** and expand the **Bind Variables** section.
3. In the **Bind Variables** section, click the **Create New Bind Variable** icon to define the bind variable.
4. In the Bind Variable dialog, enter the name and data type for the new bind variable.

Because the bind variables share the same namespace as view object attributes, specify names that don't conflict with existing view object attribute names. As

with view objects attributes, by convention bind variable names are created with an initial capital letter, but you can rename it as desired.

5. Optionally, specify a default value for the bind variable:
 - When you want the value to be determined at runtime using an expression, enter a Groovy scripting language expression, select the **Expression** value type and enter the expression in the **Value** field. Optionally, click **Edit** to open the Expression dialog. The Expression dialog gives you a larger text area to write the expression.
 - When you want to define a default value, select the **Literal** value type and enter the literal value in the **Value** field.
6. Decide on one of the following runtime usages for the bind variable:
 - When you want the value to be supplied to a SQL WHERE clause using a bind variable in the clause, select the **Required** checkbox. This ensures that a runtime exception will be thrown if the value is not supplied. For more information, see [Section 5.9.7.3, "Errors Related to Naming Bind Variables"](#).
 - When you want the value to be supplied to a view criteria using a bind variable in the view criteria, only select the **Required** checkbox when you need to reference the same bind variable in a SQL WHERE clause or when you want to use the bind variable as the assigned value of a view criteria item that is specifically defined as required by a view criteria that is applied to a view object. This ensures that the value is optional and that no runtime exception will be thrown if the bind variable is not resolved. For example, view criteria with bind variables defined can be used to create Query-by-Example search forms in the user interface. For more information, see [Section 5.10, "Working with Named View Criteria"](#)
7. Select the **Control Hints** tab and specify UI hints like **Label Text**, **Format Type**, **Format mask**, and others.

The view layer will use bind variable control hints when you build user interfaces like search pages that allow the user to enter values for the named bind variables. The **Updatable** checkbox controls whether the end user will be allowed to change the bind variable value through the user interface. If a bind variable is not updatable, then its value can only be changed programmatically by the developer.

After defining the bind variables, the next step is to reference them in the SQL statement. While SQL syntax allows bind variables to appear both in the SELECT list and in the WHERE clause, you'll typically use them in the latter context, as part of your WHERE clause. For example, [Example 5-20](#) shows the bind variables `:LowUserId` and `:HighUserId` introduced into a SQL statement created using the Query page in the overview editor for the view object.

Example 5-20 Bind Variables in the WHERE Clause of View Object SQL Statement

```
select PERSON_ID, EMAIL, FIRST_NAME, LAST_NAME
from USERS
where (upper(FIRST_NAME) like upper(:TheName) || '%'
       or upper(LAST_NAME) like upper(:TheName) || '%')
       and PERSON_ID between :LowUserId and :HighUserId
order by EMAIL
```

Notice that you reference the bind variables in the SQL statement by prefixing their name with a colon like `:TheName` or `:LowUserId`. You can reference the bind variables in any order and repeat them as many times as needed within the SQL statement.

5.9.2 What Happens When You Add Named Bind Variables

Once you've added one or more named bind variables to a view object, you gain the ability to easily see and set the values of these variables at runtime. Information about the name, type, and default value of each bind variable is saved in the view object's XML component definition file. If you have defined UI control hints for the bind variables, this information is saved in the view object's component message bundle file along with other control hints for the view object.

5.9.3 How to Test Named Bind Variables

The Business Component Browser allows you to interactively inspect and change the values of the named bind variables for any view object, which can really simplify experimenting with your application module's data model when named bind parameters are involved. For more information about editing the data model and running the Business Component Browser, see [Section 6.3, "Testing View Object Instances Using the Business Component Browser"](#).

The first time you execute a view object in the Business Component Browser to display the results in the data view page, a Bind Variables dialog will appear, as shown in [Figure 5–35](#).

The Bind Variables dialog lets you:

- View the name, as well as the default and current values, of the particular bind variable you select from the list
- Change the value of any bind variable by updating its corresponding **Value** field before clicking **OK** to set the bind variable values and execute the query
- Inspect and set the bind variables for the view object in the current data view page, using the **Edit Bind Parameters** button in the toolbar — whose icon looks like ":id"
- Verify control hints are correctly set up by showing the label text hint in the **Bind Variables** list and by formatting the **Value** attribute using the respective format mask

Figure 5–35 Setting Bind Variables in the Business Component Browser



If you defined the bind variable in the Bind Variables dialog with the **Reference** checkbox deselected (the default), you will be able to test view criteria and supply the bind variable with values as needed. Otherwise, if you selected the **Reference** checkbox, then you must supply a value for the bind variable in the Business Component Browser. The Business Component Browser will throw the same exception

seen at runtime for any view object whose SQL statement use bind variables that do not resolve with a supplied value.

5.9.4 How to Add a WHERE Clause with Named Bind Variables at Runtime

Using the view object's `setWhereClause()` method, you can add an additional filtering clause at runtime. This runtime-added WHERE clause predicate does *not* replace the design-time generated predicate, but rather further narrows the query result by adding to the existing design time WHERE clause. Whenever the dynamically added clause refers to a value that might change during the life of the application, you should use a named bind variable instead of concatenating the literal value into the WHERE clause predicate.

For example, assume you want to further filter the `UserList` view object at runtime based on the value of the `USER_ROLE` column in the table. Also assume that you plan to search sometimes for rows where `USER_ROLE = 'technician'` and other times for rows where `USER_ROLE = 'User'`. While slightly fewer lines of code, [Example 5–21](#) is not desirable because it changes the WHERE clause twice just to query two different *values* of the same `USER_ROLE` column.

Example 5–21 Incorrect Use of `setWhereClause()` Method

```
// Don't use literal strings if you plan to change the value!
vo.setWhereClause("user_role = 'technician'");
// execute the query and process the results, and then later...
vo.setWhereClause("user_role = 'user'");
```

Instead, you should add a WHERE clause predicate that references named bind variables that you define at runtime as shown in [Example 5–22](#).

Example 5–22 Correct Use of `setWhereClause()` Method and Bind Variable

```
vo.setWhereClause("user_role = :TheUserRole");
vo.defineNamedWhereClauseParam("TheUserRole", null, null);
vo.setNamedWhereClauseParam("TheUserRole", "technician");
// execute the query and process the results, and then later...
vo.setNamedWhereClauseParam("TheUserRole", "user");
```

This allows the text of the SQL statement to stay the same, regardless of the value of `USER_ROLE` you need to query on. When the query text stays the same across multiple executions, the database will return the results without having to reparse the query.

If you later need to remove the dynamically added WHERE clause and bind variable, you should do so the next time you need them to be different, just before executing the query. This will prevent the type of SQL execution error as described in [Section 5.9.7.1, "An Error Related to Clearing Bind Variables"](#). Avoid calling `removeNamedWhereClauseParam()` in your code immediately after setting the WHERE clause. For a useful helper method to assist with this removal, see [Section 5.9.7.2, "A Helper Method to Remove Named Bind Variables"](#).

An updated test client class illustrating these techniques would look like what you see in [Example 5–23](#). In this case, the functionality that loops over the results several times has been refactored into a separate `executeAndShowResults()` method. The program first adds an additional WHERE clause of `user_id = :TheUserId` and then later replaces it with a second clause of `user_role = :TheUserRole`.

Example 5–23 TestClient Program Exercising Named Bind Variable Techniques

```

package devguide.examples.client;
import oracle.jbo.ApplicationModule;
import oracle.jbo.Row;
import oracle.jbo.ViewObject;
import oracle.jbo.client.Configuration;
import oracle.jbo.domain.Number;
public class TestClient {
    public static void main(String[] args) {
        String      amDef = "devguide.examples.UserService";
        String      config = "UserServiceLocal";
        ApplicationModule am =
            Configuration.createRootApplicationModule(amDef,config);
        ViewObject vo = am.findViewObject("UserList");
        // Set the two design time named bind variables
        vo.setNamedWhereClauseParam("TheName", "alex%");
        vo.setNamedWhereClauseParam("HighUserId", new Number(315));
        executeAndShowResults(vo);
        // Add an extra where clause with a new named bind variable
        vo.setWhereClause("user_id = :TheUserId");
        vo.defineNamedWhereClauseParam("TheUserId", null, null);
        vo.setNamedWhereClauseParam("TheUserId", new Number(303));
        executeAndShowResults(vo);
        vo.removeNamedWhereClauseParam("TheUserId");
        // Add an extra where clause with a new named bind variable
        vo.setWhereClause("user_role = :TheUserRole");
        vo.defineNamedWhereClauseParam("TheUserRole", null, null);
        vo.setNamedWhereClauseParam("TheUserRole", "user");
        // Show results when :TheUserRole = 'user'
        executeAndShowResults(vo);
        vo.setNamedWhereClauseParam("TheUserRole", "technician");
        // Show results when :TheUserRole = 'technician'
        executeAndShowResults(vo);
        Configuration.releaseRootApplicationModule(am,true);
    }
    private static void executeAndShowResults(ViewObject vo) {
        System.out.println("---");
        vo.executeQuery();
        while (vo.hasNext()) {
            Row curUser = vo.next();
            System.out.println(curUser.getAttribute("UserId") + " " +
                curUser.getAttribute("Email"));
        }
    }
}

```

However, if you run this test program, you actually get a runtime error like the one shown in [Example 5–24](#).

Example 5–24 Runtime Error Resulting From a SQL Parsing Error

```

oracle.jbo.SQLStmtException: JBO-27122: SQL error during statement preparation.
Statement:
SELECT * FROM (select USER_ID, EMAIL, FIRST_NAME, LAST_NAME
from USERS
where (upper(FIRST_NAME) like upper(:TheName)||'%'
      or upper(LAST_NAME) like upper(:TheName)||'%')
      and USER_ID between :LowUserId and :HighUserId
order by EMAIL) QRSLT WHERE (user_role = :TheUserRole)
## Detail 0 ##

```

```
java.sql.SQLException: ORA-00904: "USER_ROLE": invalid identifier
```

The root cause, which appears after the ## Detail 0 ## in the stack trace, is a SQL parsing error from the database reporting that USER_ROLE column does not exist even though the USERS table definitely has a USER_ROLE column. The problem occurs due to the mechanism that view objects use by default to apply additional runtime WHERE clauses on top of read-only queries. [Section 5.9.6, "What Happens at Runtime"](#), explains a resolution for this issue.

5.9.5 How to Set Existing Bind Variable Values at Runtime

To set named bind variables at runtime, use the `setNamedWhereClauseParam()` method on the `ViewObject` interface. In JDeveloper, you can choose **Refactor > Duplicate** to create a new `TestClientBindVars` class based on the existing `TestClient.java` class as shown in [Section 6.4.2, "How to Create a Command-Line Java Test Client"](#). In the test client class, you can set the values of the bind variables using a few additional lines of code. For example, the `setNamedWhereClauseParam()` might take as arguments the bind variables `HighUserId` and `TheName` as shown in [Example 5-25](#).

Example 5-25 Setting the Value of Named Bind Variables Programmatically

```
// changed lines in TestClient class
ViewObject vo = am.findViewObject("UserList");
vo.setNamedWhereClauseParam("TheName", "alex%");
vo.setNamedWhereClauseParam("HighUserId", new Number(315));
vo.executeQuery();
// etc.
```

Running the test client class shows that your bind variables are filtering the data. For example, the resulting rows for the `setNamedWhereClauseParam()` method shown in [Example 5-25](#) may show only two matches based on the name `alex` as shown in [Example 5-26](#).

Example 5-26 Result of Bind Variables Filtering the Data in TestClient Class

```
303 ahunold
315 akhoo
```

Whenever a view object's query is executed, you can view the actual bind variable values in the runtime debug diagnostics like the sample shown in [Example 5-27](#).

Example 5-27 Debug Diagnostic Sample With Bind Variable Values

```
[256] Bind params for ViewObject: UserList
[257] Binding param "LowUserId": 0
[258] Binding param "HighUserId": 315
[259] Binding param "TheName": alex%
```

This information that can be invaluable when debugging your applications. Notice that since the code did not set the value of the `LowUserId` bind variable, it took on the default value of 0 (zero) specified at design time. Also notice that the use of the `UPPER()` function in the `WHERE` clause and around the bind variable ensured that the match using the bind variable value for `TheName` was performed case-insensitively. The sample code set the bind variable value to "alex%" with a lowercase "a", and the results show that it matched Alexander.

5.9.6 What Happens at Runtime

If you dynamically add an additional WHERE clause at runtime to a read-only view object, its query gets nested into an inline view before applying the additional WHERE clause.

For example, suppose your query was defined as shown in [Example 5–28](#).

Example 5–28 Query Specified At Design Time

```
select USER_ID, EMAIL, FIRST_NAME, LAST_NAME
from USERS
where (upper(FIRST_NAME) like upper(:TheName) || '%'
       or upper(LAST_NAME) like upper(:TheName) || '%')
       and USER_ID between :LowUserId and :HighUserId
order by EMAIL
```

At runtime, when you set an additional WHERE clause like user_role = :TheUserRole as the test program did in [Example 5–23](#), the framework nests the original query into an inline view like the sample shown in [Example 5–29](#).

Example 5–29 Runtime-Generated Query With Inline Nested Query

```
SELECT * FROM(
select USER_ID, EMAIL, FIRST_NAME, LAST_NAME
from USERS
where (upper(FIRST_NAME) like upper(:TheName) || '%'
       or upper(LAST_NAME) like upper(:TheName) || '%')
       and USER_ID between :LowUserId and :HighUserId
order by EMAIL) QRSLT
```

Then the framework adds the dynamic WHERE clause predicate at the end, so that the final query the database sees is like the sample shown in [Example 5–30](#).

Example 5–30 Runtime-Generated Query With Dynamic WHERE Clause

```
SELECT * FROM(
select USER_ID, EMAIL, FIRST_NAME, LAST_NAME
from USERS
where (upper(FIRST_NAME) like upper(:TheName) || '%'
       or upper(LAST_NAME) like upper(:TheName) || '%')
       and USER_ID between :LowUserId and :HighUserId
order by EMAIL) QRSLT
WHERE user_role = :TheUserRole
```

This query "wrapping" is necessary in the general case since the original query could be arbitrarily complex, including SQL UNION, INTERSECT, MINUS, or other operators that combine multiple queries into a single result. In those cases, simply "gluing" the additional runtime WHERE clause onto the end of the query text could produce unexpected results because, for example, it might apply only to the *last* of several UNION'ed statements. By nesting the original query verbatim into an inline view, the view object guarantees that your additional WHERE clause is correctly used to filter the results of the original query, regardless of how complex it is. The consequence (that results in an ORA-904 error) is that the dynamically added WHERE clause can refer only to columns that have been selected in the original query.

The simplest solution is to add the dynamic query column names to the end of the query's SELECT list on the Edit Query dialog (click the **Edit** icon on the Query page of the overview editor for the view object). Just adding the new column name at the end of the existing SELECT list — of course, preceded by a comma — is enough to prevent

the ORA-904 error: JDeveloper will automatically keep your view object's attribute list in sync with the query statement. Alternatively, [Section 35.3.3.7, "Disabling the Use of Inline View Wrapping at Runtime"](#) explains how to disable this query nesting when you don't require it.

The modified test client program in [Example 5–23](#) now produces the expected results shown in [Example 5–31](#).

Example 5–31 Named Bind Variables Resulting From Corrected TestClient

```
---
303 ahunold
315 akhoo
---
303 ahunold
---
315 akhoo
---
303 ahunold
```

5.9.7 What You May Need to Know About Named Bind Variables

There are several things you may need to know about named bind variables, including the runtime errors that are displayed when bind variables have mismatched names and the default value for bind variables.

5.9.7.1 An Error Related to Clearing Bind Variables

You need to ensure that your application handles changing the value of bind variables properly for use with activation and passivation of the view object instance settings at runtime. For example, before you deploy the application, you will want to stress-test your application in JDeveloper by disabling application module pooling, as described in [Section 36.10, "Testing to Ensure Your Application Module is Activation-Safe"](#).

Following the instructions in that section effectively simulates the way your application will manage the passivation store when you eventually deploy the application.

When the application reactivates the pending state from the passivation store upon subsequent requests during the same user session, the application will attempt to set the values of any dynamically added named WHERE clause bind variables. Changing the values to null before passivation take places will prevent the bind variable values from matching the last time the view object was executed and the following error will occur during activation:

```
(oracle.jbo.SQLStmtException) JBO-27122: SQL error during statement preparation.
(java.sql.SQLException) Attempt to set a parameter name that does not occur in
SQL: 1
```

Do not change the value of the bind variables (or other view object instance settings) just after executing the view object. Rather, if you will not be re-executing the view object again during the same block of code (and therefore during the same HTTP request), you should defer changing the bind variable values for the view object instance until the next time you need them to change, just before executing the query.

Therefore, do not follow this pattern:

1. (Request begins and application module is acquired)
2. Calling `setWhereClause()` on a view object instance that references n bind variables

3. Calling `setWhereClauseParam()` to set the n values for those n bind variables
4. Calling `executeQuery()`
5. Calling `setWhereClause(null)` to clear WHERE clause
6. Calling `setWhereClauseParam(null)` to clear the WHERE clause bind variables
7. (Application module is released)

Instead, use the follow pattern:

1. (Request begins and application module is acquired)
2. Call `setWhereClause(null)` to clear WHERE clause
3. Call `setWhereClauseParam(null)` to clear the WHERE clause bind variables
4. Call `setWhereClause()` that references n bind variables
5. Calling `setWhereClauseParam()` to set the n values for those n bind variables
6. Calling `executeQuery()`
7. (Application module is released)

5.9.7.2 A Helper Method to Remove Named Bind Variables

The helper method `clearWhereState()` that you can add to your `ViewObjectImpl` framework ensures that declaratively defined bind variables are not removed. [Example 5–32](#) shows the use of `clearWhereState()` to safely remove named bind variables that have been added to the view instance at runtime.

Example 5–32 Helper Method to Clear Named Bind Variables Values Programmatically

```
protected void clearWhereState() {
    ViewDefImpl viewDef = getViewDef();
    Variable[] viewInstanceVars = null;
    VariableManager viewInstanceVarMgr = ensureVariableManager();
    if (viewInstanceVarMgr != null) {
        viewInstanceVars = viewInstanceVarMgr.getVariablesOfKind
            (Variable.VAR_KIND_WHERE_CLAUSE_PARAM);
        if (viewInstanceVars != null) {
            for (Variable v: viewInstanceVars) {
                // only remove the variable if its not on the view def.
                if (!hasViewDefVariableNamed(v.getName())) {
                    removeNamedWhereClauseParam(v.getName());
                }
            }
        }
    }
    getDefaultRowSet().setExecuteParameters(null, null, true);
    setWhereClause(null);
    getDefaultRowSet().setWhereClauseParams(null);
}
private boolean hasViewDefVariableNamed(String name) {
    boolean ret = false;
    VariableManager viewDefVarMgr = getViewDef().ensureVariableManager();
    if (viewDefVarMgr != null) {
        try {
            ret = viewDefVarMgr.findVariable(name) != null;
        }
        catch (NoDefException ex) {
            // ignore
        }
    }
}
```

```

    }
    return ret;
}

```

5.9.7.3 Errors Related to Naming Bind Variables

You need to ensure that the list of named bind variables that you reference in your SQL statement matches the list of named bind variables that you've defined in the Bind Variables section of the overview editor's query page for the view object. Failure to have these two agree correctly can result in one of the following two errors at runtime.

If you use a named bind variable in your SQL statement but have not defined it, you'll receive an error like this:

```
(oracle.jbo.SQLStmtException) JBO-27122: SQL error during statement preparation.
## Detail 0 ##
(java.sql.SQLException) Missing IN or OUT parameter at index:: 1
```

On the other hand, if you have defined a named bind variable, but then forgotten to reference it or mistyped its name in the SQL, then you will see an error like this:

```
oracle.jbo.SQLStmtException: JBO-27122: SQL error during statement preparation.
## Detail 0 ##
java.sql.SQLException: Attempt to set a parameter name that does not occur in the
SQL: LowUserId
```

To resolve either of these errors, double-check that the list of named bind variables in the SQL matches the list of named bind variables in the Bind Variables section of the overview editor's Query page for the view object. Additionally, open the Bind Variables dialog for the bind variable and verify that the **Reference** checkbox is not still deselected (the default). To use the bind variable in a SQL statement, you must select the **Reference** checkbox.

5.9.7.4 Default Value of NULL for Bind Variables

If you do not supply a default value for your named bind variable, it defaults to the NULL value at runtime. This means that if you have a WHERE clause like:

```
USER_ID = :TheUserId
```

and you do not provide a default value for the TheUserId bind variable, it will default to having a NULL value and cause the query to return no rows. Where it makes sense for your application, you can leverage SQL functions like NVL(), CASE, DECODE(), or others to handle the situation as you require. For example, the following WHERE clause fragment allows the view object query to match any name if the value of :TheName is null.

```
upper(FIRST_NAME) like upper(:TheName) || '%'
```

5.10 Working with Named View Criteria

A view criteria you define lets you specify filter information for the rows of a view object collection. The view criteria object is a row set of one or more view criteria rows, whose attributes mirror those in the view object. The view criteria definition comprises query conditions that augment the WHERE clause of the target view object. Query conditions that you specify apply to the individual attributes of the target view object.

The key difference between a view object row of query results and a view criteria row is that the data type of each attribute in the view criteria row is `String`. This key difference supports Query-by-Example operators and therefore allows the user to enter conditions such as "`OrderId > 304`", for example.

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

Additionally, view criteria have full API support, and it is therefore possible to create and apply view criteria to view objects programmatically.

5.10.1 How to Create Named View Criteria Declaratively

You create named view criteria definitions when you need to filter individual view object results. View criteria that you define at design time can participate in these scenarios where filtering results is desired at runtime:

- Supporting Query-by-Example search forms that allow the end user to supply values for attributes of the target view object.

For example, the end user might input the value of a customer name and the date to filter the results in a web page that displays the rows of the `CustomerOrders` view object. The web page designer will see the named view criteria in the JDeveloper Data Controls panel and, from them, easily create a search form. For more information about utilizing the named view criteria in the Data Controls panel, see [Section 25.2, "Creating Query Search Forms"](#).

- Filtering the list of values (LOV) that allow the end user may select from one attribute list (displayed in the UI as an LOV component).

The web page designer will see the attributes of the view object in the JDeveloper Data Controls panel and, from them, easily create LOV controls. For more information about utilizing LOV-enabled attributes in the Data Controls panel, see [Section 23.1, "Creating a Selection List"](#).

- Validating attribute values using a view accessor with a view criteria applied to filter the view accessor results.

For more information about create view accessor validators, see [Section 10.4.2, "How to Validate Against a View Accessor"](#).

- Creating the application module's data model from a single view object definition with a unique view criteria applied for each view instance.

The single view object query modified by view criteria is useful with look up data that must be shared across the application. In this case, a base view object definition queries the lookup table in the database and the view criteria set the lookup table's `TYPE` column to define application-specific views. To define view instances in the data model using the view criteria you create for a base view object definition, see [Section 10.3.3, "How to Define the WHERE Clause of the Lookup View Object Using View Criteria"](#).

To define view criteria for the view object you wish to filter, you open the view object in the overview editor and use the **View Criteria** section of the Query page. A dedicated editor that you open from the **View Criteria** section helps you to build a `WHERE` clause using attribute names instead of the target view object's corresponding SQL column names. You may define multiple named view criteria for each view object.

Each view criteria definition consists of the following elements:

- One or more view criteria rows consisting of an arbitrary number of view criteria groups or an arbitrary number of references to another named view criteria already defined for the current view object.
- Optional view criteria groups consisting of an arbitrary number of view criteria items.
- View criteria items consisting of an attribute name, an attribute-appropriate operator, and an operand. Operands can be a literal value when the filter value is defined or a bind variable that can optionally utilize a scripting expression that includes dot notation access to attribute property values.

Expressions are based on the Groovy scripting language, as described in [Section 3.6, "Overview of Groovy Support"](#).

When you define a view criteria, you control the source of the filtered results. You can limit the results of the filtered view object to:

- Just the database table specified by the view object
- Just the in-memory results of the view object query
- Both the database and the in-memory results of the view object query.

Filtering on both database tables and the view object's in-memory results allows you to filter rows that were created in the transaction but not yet committed to the database.

View criteria expressions you construct in the Edit View Criteria dialog use logical conjunctions to specify how to join the selected criteria item or criteria group with the previous item or group in the expression:

- AND conjunctions specify that the query results meet both joined conditions. This is the default for each view criteria item you add.
- OR conjunctions specify that the query results meet either or both joined conditions. This is the default for view criteria groups.

Additionally, you may create nested view criteria when you want to filter rows in the current view object based on criteria applied to view-linked detail views. A nested view criteria group consists of an arbitrary number of nested view criteria items. You can use nested view criteria when you want to have more controls over the logical conjunctions among the various view criteria items. The nested criteria place restrictions on the rows that satisfy the criteria under the nested criteria's parent view criteria group. For example, you might want to query both a list of employees with (Salary > 3000) and belonging to (DeptNo = 10 or DeptNo = 20). You can define a view criteria with the first group with one item for (Salary > 3000) and a nested view criteria with the second group with two items DeptNo = 10 and DeptNo = 20.

Note: Query search forms do not support view criteria defined by nested expressions (multiple groups of view criteria items). If you intend the UI designer to be able to create a query search form based on the view criteria, do not create a view criteria with multiple groups in its expression.

To define a named view criteria:

1. In the Application Navigator, double-click the view object for which you want to create the named view criteria.

2. In the overview editor navigation list, select **Query** and expand the **View Criteria** section.
3. In the View Criteria section, click the **Create New View Criteria** icon.
4. In the Edit View Criteria dialog, enter the name of the view criteria to identify its usage in your application.
5. In the **Query Execution Mode** dropdown list, decide how you want the view criteria to filter the view object query results.

You can limit the view criteria to filter the database table specified by the view object query, the in memory row set produced by the query, or both the database table and the in-memory results.

Choosing **Both** may be appropriate for situations where you want to include rows created as a result of enforced association consistency. In this case, in-memory filtering is performed after the initial fetch.

6. Click one of these **Add** buttons to define the view criteria.
 - **Add Item** to add a single criteria item. The editor will add the item to the hierarchy beneath the current group or view criteria selection. By default each time you add an item, the editor will choose the next attribute to define the criteria item. You can change the attribute to any attribute that the view object query defines.
 - **Add Group** to add a new group that will compose criteria items that you intend to add to it. When you add a new group, the editor inserts the **OR** conjunction into the hierarchy. You can change the conjunction as desired.
 - **Add Criteria** to add a view criteria that you intend to define. This selection is an alternative to adding a named criteria that already exists in the view object definition. When you add a new view criteria, the editor inserts the **AND** conjunction into the hierarchy. You can change the conjunction as desired. Each time you add another view criteria, the editor nests the new view criteria beneath the current view criteria selection in the hierarchy. The root node of the hierarchy defines the named view criteria that you are currently editing.
 - **Add Named Criteria** to add a view criteria that the view object defines. The named criteria must appear in the overview editor for the view object you are defining the view criteria.

Search forms that the UI designer will create from view criteria are restricted to the criteria of the first group. Subsequent groups will not produce a runtime error but their criteria items will be ignored by the search form.

7. Select a view criteria item node in the view criteria hierarchy and define the added node in the **Criteria Item** section.
 - Choose the desired **Attribute** for the criteria item. By default the editor adds the first one in the list.

Optionally, you can add a nested view criteria inline when a view link exists for the current view object you are editing. The destination view object name will appear in the **Attribute** dropdown list. Selecting a view object lets you filter the view criteria based on view criteria items for the nested view criteria based on a view link relationship. For example, **AddressVO** is linked to the **PaymentOptionsVO** and a view criteria definition for **PaymentOptionsVO** will display the destination view object **AddressVO**. You could define the nested view criteria to filter payment options based on the **CountryId**.

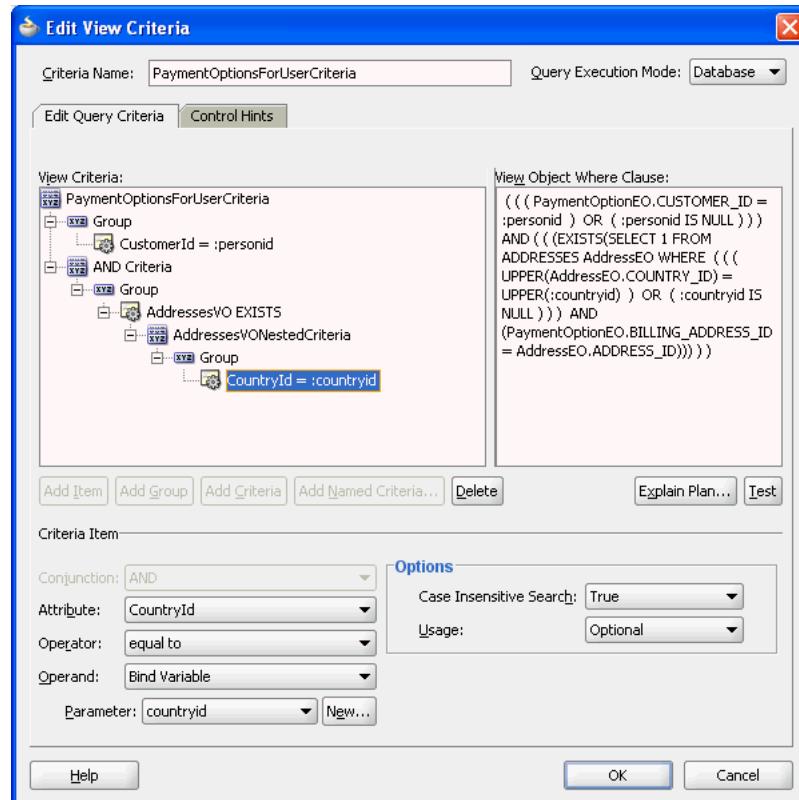
attribute of the current customer, as specified by the `CustomerId` criteria item, as shown in [Figure 5–36](#).

- Choose the desired **Operator**.
JDeveloper does not support the `IN` operator. However, you can create a view criteria with the `IN` operator using the API, as described in [5.10.4 , "How to Create View Criteria Programmatically"](#).
- 8. Choose the desired **Operand** for the view criteria item selection.
 - Select **Literal** when you want to supply a value for the attribute or when you want to define a default value for a user-specified search field for a Query-by-Example search form. When the view criteria defines a query search form for the user interface, you may leave the **Value** field empty. In this case, the user will supply the value. You may also provide a value that will act as a search field default value that the user will be able to override. The value you supply in the **Value** field can include wildcard characters * or %.
 - Select **Bind Variable** when you want the value to be determined at runtime using a bind variable. If the variable was already defined for the view object, select it from the **Parameters** dropdown list. Otherwise, click **New** to display the Bind Variable dialog that lets you create a new bind variable on the view object. For more information about creating bind variables, see [Section 5.9.1, "How to Add Bind Variables to a View Object Definition"](#).
When you define bind variables on the view object for use by the view criteria, you must specify that the variable is not required by the SQL query that the view object defines. To do this, deselect the **Required** checkbox in the Bind Variables dialog, as explained in [Section 5.9.1, "How to Add Bind Variables to a View Object Definition"](#)
- 9. For each item, group, or nested view criteria that you define, optionally change the default conjunction to specify how the selection should be joined.
 - **AND** conjunction specifies that the query results meet both joined conditions. This is the default for each view criteria item or view nested view criteria that you add.
 - **OR** conjunction specifies that the query results meet either or both joined conditions. This is the default for view criteria groups.
- 10. Verify that the view criteria definition is valid by doing one of the following:
 - Click **Explain Plan** to visually inspect the view criteria's generated `WHERE` clause.
 - Click **Test** to allow JDeveloper to verify that the `WHERE` clause is valid.
- 11. In the **Case Insensitive Search** dropdown list, select **True** when you do not want the value supplied for an attribute to be filtered based on the case of the value.
The attribute value can be a literal value that you define or a runtime parameter that the end user supplies. This option is enabled for attributes of type String only. The default **False** enables case sensitive searches.
- 12. In the **Usage** dropdown list, decide whether the view criteria item is a required or an optional part of the attribute value comparison in the generated `WHERE` clause.
The usage **Selectively Required** means that the `WHERE` clause will ignore the view criteria item at runtime if no value is supplied and there exists at least one criteria item at the same level that has a criteria value. Otherwise, an exception is thrown. The usage **Optional** means the view criteria is added to the `WHERE` clause only if

the value is non-NULL. The default **Optional** for each new view criteria item means no exception will be generated for null values. The usage **Required** means that the WHERE clause will fail to execute and an exception will be thrown when no value is supplied for the criteria item.

13. Click **OK**.

Figure 5–36 Edit View Criteria Dialog with View Criteria Specified



5.10.2 How to Set User Interface Hints on View Criteria

Named view criteria that you create for view object collections can be used by the web page designer to create Query-by-Example search forms. Web page designers select your named view criteria from the JDeveloper Data Controls panel to create search forms for the Fusion web application. In the web page, the search form utilizes an ADF Faces query search component that will be bound initially to the named view criteria selected in the Data Controls panel. At runtime, the end user may select among all other named view criteria that appear in the Data Controls panel. Named view criteria that the end user can select in a search form are known as *developer-seeded searches*. The query component automatically displays these seeded searches in its **Saved Search** dropdown list. For more information about creating search forms and using the ADF query search component, see [Section 25.2, "Creating Query Search Forms"](#).

Note: By default, any named view criteria you create in the Edit View Criteria dialog will appear in the Data Controls panel. As long as the **Show In List** option appears selected in the Control Hints page of the Edit View Criteria dialog, JDeveloper assumes that the named view criteria should be available as a developer-seeded search. When you want to create a named view criteria that you do not want the end user to see in search forms, deselect the **Show In List** option in the dialog. For example, you might create a named view criteria only for an LOV-enabled attribute and so you would need to deselect **Show In List**.

Because developer-seeded searches are created in the data model project, the Control Hints page of the Edit View Criteria dialog lets you specify the default properties for the query component's runtime usage of individual named view criteria. At runtime, the query component's behavior will conform to the selections you make for the following seeded search properties:

Query Automatically: Select when you want the query associated with the named view criteria to be executed and the results displayed in the web page. Any developer-seeded search with this option enabled will automatically be executed when the end user selects it from the query component's **Saved Search** list. Deselect when the web page designer prefers not to display the results until the end user submits the search criteria values on the form. By default, this option is disabled for a view criteria you define in the Edit View Criteria dialog.

Search Region Mode: Select the mode that you want the query component to display the seeded search as. The **Basic** mode has all features of **Advanced** mode, except that it does not allow the end user to dynamically modify the displayed search criteria fields. The default is **Basic** mode for a view criteria you define in the Edit View Criteria dialog.

Show Match All and Match Any: Select to allow the query component to display the **Match All** and **Match Any** radio selection buttons to the end user. When these buttons are present, the end user can use them to modify the search to return matches for all criteria or any one criteria. This is equivalent to enforcing AND (match all) or OR (match any) conjunctions between view criteria items. Deselect when you want the view criteria to be executed using the conjunctions it defines. In this case, the query component will not display the radio selection buttons.

Show Operators: Determine how you want the query component to display the operator selection field for the view criteria items to the end user. For example, select **Always** when you want to allow the end user to customize the operators for criteria items (in either basic or advanced modes) or select **Never** when you want the view criteria to be executed using the operators it defines.

Show In List: Select to ensure that the view criteria is defined as a developer-seeded query. Deselect when the named view criteria you are defining is not to be used by the query search component to display a search form. Your selection determines whether the named view criteria will appear in the query search component's **Saved Search** dropdown list of available seeded searches. By default, this option is enabled for a view criteria you define in the Edit View Criteria dialog.

Display Name: Enter the name of the seeded search that you want to appear in the query component's **Saved Search** dropdown list. The name you enter will be the name by which the end user identifies the seeded search. When no display name is entered, the view criteria name displayed in the Edit View Criteria dialog will be used by default.

Rendered Mode: Select individual view criteria items from the view criteria tree component and choose whether you want the selected item to appear in the search form when the end user toggles the query component between basic mode and advanced mode. The default for every view criteria item is **All**. The default mode permits the query component to render an item in either basic or advanced mode. By changing the **Rendered Mode** setting for individual view criteria items, you can customize the search form's appearance at runtime. For example, you may want basic mode to display a simplified search form to the end user, reserving advanced mode for displaying a search form with the full set of view criteria items. In this case, you would select **Advanced** for the view criteria item that you do not want displayed in the query component's basic mode. In contrast, when you want the selected view criteria item to be rendered only in basic mode, select **Basic**. Set any item that you do not want the search form to render in either basic or advanced mode to **Never**.

Note: When your view criteria includes an item that should not be exposed to the user, use the **Rendered Mode** setting **Never** to prevent it from appearing in the search form. For example, a view criteria may be created to search for products in the logged-in customer's cart; however, you may want to prevent the user from changing the customer ID to display another customer's cart contents. In this scenario, the view criteria item corresponding to the customer ID would be set to the current customer ID using a named bind variable. Although the bind variable definition might specify the variable as not required and not updatable, with the Control Hint property **Display** set to **Hide**, only the **Rendered Mode** setting determines whether or not the search form displays the value.

To create a seeded search for use by the ADF query search component, you select **Show In List** in the Control Hints page of the Edit View Criteria dialog. You deselect **Show In List** when you do not want the end user to see the view criteria in their search form.

To customize a named view criteria for the user interface:

1. In the Application Navigator, double-click the view object that defines the named view criteria you want to use as a seeded search.
2. In the overview editor navigation list, select **Query** and expand the **View Criteria** section.
3. In the **View Criteria** section, double-click the named view criteria that you want to allow in seeded searches.
4. On the Control Hints page of the Edit View Criteria dialog, ensure that **Show In List** is selected.

This selection determines whether or not the query component will display the seeded search in its **Saved Search** dropdown list.

5. Enter a user-friendly display name for the seeded search to be added to the query component **Saved Search** dropdown list.

When left empty, the view criteria name displayed in the Edit View Criteria dialog will be used by the query component.

6. Optionally, enable **Query Automatically** when you want the query component to automatically display the search results whenever the end user selects the seeded search from the **Saved Search** dropdown list.

By default, no search results will be displayed.

7. Optionally, apply **Criteria Item Control Hints** to customize whether the query component renders individual criteria items when the end user toggles the search from between basic and advanced mode.

By default, all view criteria items defined by the seeded search will be displayed in either mode.

If a rendered criteria item is of type Date, you must also define Control Hints for the corresponding view object attribute. Set the view object attribute's Format Type hint to **Simple Date** and set the Format Mask to an appropriate value, as described in [Section 5.12.1, "How to Add Attribute Control Hints"](#). This will allow the search form to accept date values.

8. Click **OK**.

5.10.3 How to Test View Criteria Using the Business Component Browser

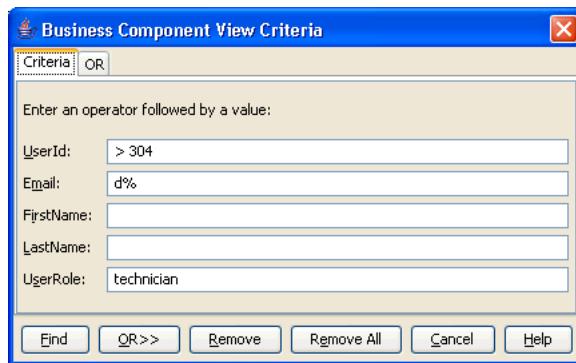
To test the view criteria you added to a view object, use the Business Component Browser, which is accessible from the Application Navigator.

The Business Component Browser, for any view object instance that you browse, lets you bring up the Business Components View Criteria dialog, as shown in [Figure 5–37](#). The dialog allows you to create a view criteria comprising one or more view criteria rows.

To apply criteria attributes from a single view criteria row, click the **Specify View Criteria** toolbar icon in the browser and enter Query-by-Example criteria in the desired fields, then click **Find**.

To test view criteria using the Business Component Browser:

1. In the Application Navigator, expand the project containing the desired application module and view objects.
2. Right-click the application module and choose **Run**.
3. In the Business Component Browser, click the **Specify View Criteria** toolbar icon and enter Query-by-Example criteria in the desired fields, then click **Find**.
4. In the Business Component Browser, click the **Specify View Criteria** toolbar icon.
5. In the Business Components View Criteria dialog, perform one of the following tasks:
 - To apply criteria attributes from a single view criteria row, enter the desired values for the view criteria and click **Find**. For example, [Figure 5–37](#) shows the filter to return all users who are members of the technician role and who possess a user ID greater than 304 and an email address that begins with the letter "d".
 - To add additional view criteria rows, click the **OR** tab and use the additional tabs that appear to switch between pages, each representing a distinct view criteria row. When you click **Find**, the Business Component Browser will create and apply the view criteria to filter the result.

Figure 5–37 Business Components View Criteria Dialog

5.10.4 How to Create View Criteria Programmatically

To create and apply a view criteria at runtime, follow the steps illustrated in the `TestClientViewCriteria` class in [Example 5–33](#) to call:

1. `createViewCriteria()` on the view object, to be filtered to create an empty view criteria row set
2. `createViewCriteriaRow()` on the view criteria, to create one or more empty view criteria rows
3. `setAttribute()` on the view criteria rows and items, to set attribute name, comparison operator, and value to filter on respectively
Alternatively, use `ensureCriteriaItem()`, `setOperator()`, and `setValue()` on the view criteria rows and items, to set attribute name, comparison operator, and value to filter on respectively.
4. `add()` on the view criteria, to add the view criteria rows to the view criteria row set
5. `applyViewCriteria()`, to apply the view criteria to the view object
6. `executeQuery()` on the view criteria, to execute the query with the applied filter criteria

The last step to execute the query is important, since a newly applied view criteria is applied to the view object's SQL query only at its next execution.

Example 5–33 Creating and Applying a View Criteria

```
package devguide.examples.readonlyvo.client;

import oracle.jbo.ApplicationModule;
import oracle.jbo.Row;
import oracle.jbo.ViewCriteria;
import oracle.jbo.ViewCriteriaRow;
import oracle.jbo.ViewObject;
import oracle.jbo.client.Configuration;

public class TestClientViewCriteria {
    public static void main(String[] args) {
        String amDef = "devguide.examples.readonlyvo.PersonService";
        String config = "PersonServiceLocal";
        ApplicationModule am =
            Configuration.createRootApplicationModule(amDef, config);
        ViewObject vo = am.findViewObject("PersonList");
    }
}
```

```

// Work with your appmodule and view object here
Configuration.releaseRootApplicationModule(am, true);
// 1. Create a view criteria rowset for this view object
ViewCriteria vc = vo.createViewCriteria();
// 2. Use the view criteria to create one or more view criteria rows
ViewCriteriaRow vcr1 = vc.createViewCriteriaRow();
ViewCriteriaRow vcr2 = vc.createViewCriteriaRow();
// 3. Set attribute values to filter on in appropriate view criteria rows
vcr1.setAttribute("PersonId", "> 200");
vcr1.setAttribute("Email", "d%");
vcr1.setAttribute("PersonTypeCode", "STAFF");
vcr2.setAttribute("PersonId", "IN (204,206)");
vcr2.setAttribute("LastName", "Hemant");
// 4. Add the view criteria rows to the view criteria rowset
vc.add(vcr1);
vc.add(vcr2);
// 5. Apply the view criteria to the view object
vo.applyViewCriteria(vc);
// 6. Execute the query
vo.executeQuery();
while (vo.hasNext()) {
    Row curPerson = vo.next();
    System.out.println(curPerson.getAttribute("PersonId") + " " +
        curPerson.getAttribute("Email"));
}
}
}

```

Running the `TestClientViewCriteria` example in [Example 5-33](#) produces the output:

```

305 daustin
307 dlorentz
324 hbaer

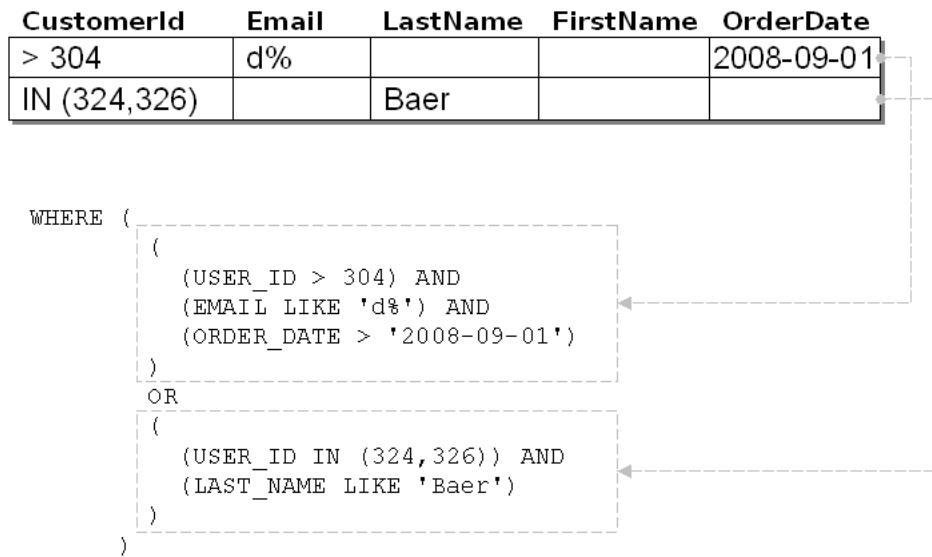
```

5.10.5 What Happens at Runtime

When you apply a view criteria containing one or more view criteria rows to a view object, the next time it is executed it augments its SQL query with an additional WHERE clause predicate corresponding to the Query-by-Example criteria that you've populated in the view criteria rows. As shown in [Figure 5-38](#), when you apply a view criteria containing multiple view criteria rows, the view object augments its design time WHERE clause by adding an additional runtime WHERE clause based on the non-null example criteria attributes in each view criteria row.

A corollary of the view criteria feature is that each time you apply a new view criteria (or remove an existing one), the text of the view object's SQL query is effectively changed. Changing the SQL query causes the database to reparse the statement the next time it is executed. You can eliminate the reparsing and improve the performance of a view criteria as described in [Section 5.10.7, "What You May Need to Know About Query-By-Example Criteria"](#).

Figure 5–38 View Object Automatically Translates View Criteria Rows into Additional Runtime WHERE Filter



5.10.6 What You May Need to Know About the View Criteria API

When you need to perform tasks that the Edit View Criteria dialog does not support, review the View Criteria API. For example, programmatically, you can alter compound search conditions using multiple view criteria rows, search for a row whose attribute value is NULL, search case insensitively, and clear view criteria in effect.

5.10.6.1 Referencing Attribute Names in View Criteria

The `setWhereClause()` method allows you to add a dynamic WHERE clause to a view object, as described in [Section 6.4.1, "ViewObject Interface Methods for Working with the View Object's Default RowSet"](#). You can also use `setWhereClause()` to pass a string that contains literal database column names like this:

```
vo.setWhereClause("LAST_NAME LIKE UPPER(:NameToFind)");
```

In contrast, when you use the view criteria mechanism, shown in [Example 5–33](#), you must reference the view object attribute name instead like this:

```
ViewCriteriaItem vc_item1 = vc_row1.ensureCriteriaItem("UserId");
vc_item1.setOperator(">");
vc_item1.setValue("304");
```

The view criteria rows are then translated by the view object into corresponding WHERE clause predicates that reference the corresponding column names.

5.10.6.2 Referencing Bind Variables in View Criteria

When you want to set the value of a view criteria item to a bind variable, use `setIsBindVarValue(true)` like this.

```
ViewCriteriaItem vc_item1 = vc_row1.ensureCriteriaItem("UserId");
vc_item1.setIsBindVarValue(true);
vc_item1.setValue(":VariableName");
```

5.10.6.3 Altering Compound Search Conditions Using Multiple View Criteria Rows

When you add multiple view criteria rows, you can call the `setConjunction()` method on a view criteria row to alter the conjunction used between the predicate corresponding to that row and the one for the previous view criteria row. The legal constants to pass as an argument are:

- `ViewCriteriaRow.VCROW_CONJ_AND`
- `ViewCriteriaRow.VCROW_CONJ_NOT`
- `ViewCriteriaRow.VCROW_CONJ_OR` (*default*)

The NOT value can be combined with AND or OR to create filter criteria like:

```
( PredicateForViewCriteriaRow1 ) AND (NOT ( PredicateForViewCriteriaRow2 ) )
```

or

```
( PredicateForViewCriteriaRow1 ) OR (NOT ( PredicateForViewCriteriaRow2 ) )
```

The syntax to achieve these requires using Java's bitwise OR operator like this:

```
vcr2.setConjunction(ViewCriteriaRow.VCROW_CONJ_AND | ViewCriteriaRow.VCROW_CONJ_NOT);
```

5.10.6.4 Searching for a Row Whose Attribute Value Is NULL Value

To search for a row containing a NULL value in a column, populate a corresponding view criteria row attribute with the value "IS NULL" or use `ViewCriteriaItem.setOperator("ISBLANK")`.

5.10.6.5 Searching Case-Insensitive

To search case-insensitively, call `setUpperColumns(true)` on the view criteria row to which you want the case-insensitivity to apply. This affects the WHERE clause predicate generated for String-valued attributes in the view object to use `UPPER(COLUMN_NAME)` instead of `COLUMN_NAME` in the predicate. Note that the value of the supplied view criteria row attributes for these String-valued attributes must be uppercase or the predicate won't match. In addition to the predicate, it also possible to use `UPPER()` on the value. For example, you can set `UPPER(ename) = UPPER("scott")`.

5.10.6.6 Clearing View Criteria in Effect

To clear any view criteria in effect, you can call `getViewCriteria()` on a view object and then delete all the view criteria rows from it using the `remove()` method, passing the zero-based index of the criteria row you want to remove. If you don't plan to add back other view criteria rows, you can also clear all the view criteria in effect by simply calling `applyViewCriteria(null)` on the view object.

5.10.7 What You May Need to Know About Query-By-Example Criteria

For performance reasons, you want to avoid setting a bind parameter as the value of a view criteria item in these two cases:

- In the specialized case where the value of a view criteria item is defined as selectively required and the value changes from non-NUL to NULL.

In this case, the SQL statement for the view criteria will be regenerated each time the value changes from non-NUL to NULL.

- In the case where the value of the view criteria item is optional and that item references an attribute for an indexed column.

In the case of optional view criteria items, an additional SQL clause `OR (:Variable IS NULL)` is generated, and the clause does not support using column indices.

In either of the following cases, you will get better performance by using a view object whose WHERE clause contains the named bind variables, as described in [Section 5.9.1, "How to Add Bind Variables to a View Object Definition"](#). In contrast to the view criteria filtering feature, when you use named bind variables, you can change the *values* of the search criteria without changing the *text* of the view object's SQL statement each time those values change.

5.11 Working with List of Values (LOV) in View Object Attributes

Edit forms displayed in the user interface portion of your application can utilize LOV-enabled attributes that you define in the data model project to predetermine a list of values for individual input fields. When the user submits the form with their selected values, ADF data bindings in the ADF Model layer update the value on the view object attributes corresponding to the databound fields. To facilitate this common design task, ADF Business Components provides declarative support to specify the LOV usage in the user interface.

Defining an LOV for attributes of a view object in the data model project greatly simplifies the task of working with list controls in the user interface. Because you define the LOV on the individual attributes of the view object, you can customize the LOV usage for an attribute once and expect to see the list component in the form wherever the attribute appears.

Note: In order for the LOV to appear in the UI, the LOV usage must exist before the user interface designer creates the databound form.

Defining an LOV usage for an attribute referenced by an existing form will not change the component that the form displays to an LOV.

You can define an LOV for any view object attribute that you anticipate the user interface will display as a selection list. The characteristics of the attribute's LOV definition depend on the requirements of the user interface. The information you gather from the user interface designer will determine the best solution. For example, you might define LOV attributes in the following cases:

- When you need to display attribute values resulting from a view object query against a business domain object.
For example, define LOV attributes to display the list of suppliers in a purchase order form.
- When you want to display attribute values resulting from a view object query that you wish to filter using a parameter value from any attribute of the LOV attribute's current row.

For example, define LOV attributes to display the list of supplier addresses in a purchase order form but limit the addresses list based on the current supplier.

If you wish, you can enable a second LOV to drive the value of the parameter based on a user selection. For example, you can let the user select the current supplier to drive the supplier addresses list. In this case, the two LOVs are known as a *cascading list*.

Before you can define the LOV attribute, you must create a data source view object in your data model project that queries the eligible rows for the attribute value you want the LOV to display. After this, you work entirely on the base view object to define the LOV. The base view object is the one that contains the primary data for display in the user interface. The LOV usage will define the following additional view object metadata:

- A view accessor to access the data source for the LOV attribute. The view accessor is the ADF Business Components mechanism that lets you obtain the full list of possible values from the row set of the data source view object.
- Optionally, supplemental values that the data source may return to attributes of the base view object other than the data source attribute for which the list is defined.
- User interface hints, including the type of list component to display, attributes to display from the current row when multiple display attributes are desirable, and a few options specific to the choice list component.

The general process for defining the LOV-enabled attribute relies on the Edit Attribute dialog that you display for the base view object attribute.

To define the LOV-enabled attribute, follow this general process:

1. Select the **Enable List of Values** option.
2. Create a new view accessor definition to point to the data source view object or select an existing view accessor that the base view object already defines.
Optionally, you can filter the view accessor by creating a view criteria using a bind variable that obtains its value from any attribute of base view object's current row.
3. Select the list attribute from the view accessor's data source view object.
This maps the attribute you select to the current attribute of the base view object.
4. Optionally, select list return values to map any supplemental values that your list returns to the base view object.
5. Select user interface hints to specify the list's display features.
6. Save the attribute changes.

Note: If you create a view criteria to filter the data source view object, you may also set an LOV on the attribute of the base view object that you use to supply the value for the view criteria bind parameter. You set cascading LOV lists when you want the user's selection of one attribute to drive the options displayed in a second attribute's list.

Once you create the LOV-enabled attribute, the user interface designer can create the list component in the web page by dragging the LOV-enabled attribute's collection from the Data Controls panel. For further information about creating a web page that display the list, see [Chapter 23, "Creating Databound Selection Lists and Shuttles"](#). Specifically, for more information about working with LOV-enabled attributes in the web page, see [Section 23.1.2, "How to Create a Model-Driven List"](#).

5.11.1 How to Define a Single LOV-Enabled View Object Attribute

When an edit form needs to display a list values that is not dependent on another selection in the edit form, you can define a view accessor to point to the list data source. For example, assume that a purchase order form contains a field that requires the user to select the order item's supplier. In this example, you would first create a view accessor that points to the data source view object (`SuppliersView`). You would then set the LOV on the `SupplierDesc` attribute of the base view object (`PurchaseOrdersView`). Finally, you would reference that view accessor from the LOV-enabled attribute (`SupplierDesc`) of the base view object and select the data source attribute (`SupplierDesc`).

Use will use the Edit Attribute dialog to define an LOV-enabled attribute for the base view object. The dialog lets you select an existing view accessor or create a new one to save with the LOV-attribute definition.

To define an LOV that displays values from a view object attribute:

1. In the Application Navigator, double-click the view object that contains the attribute you wish to enable as an LOV.
2. In the overview editor navigation list, select **Attributes** and double-click the attribute that is to display the LOV.

Alternatively, you can expand the **List of Values** section of the overview editor and click the **Add** button. Use the List of Values dialog to create the LOV on the attribute you have currently selected in the attribute list of the overview editor.

3. In the Edit Attribute dialog, select **List of Values** from the navigation list and select **Enable List of Values**.

JDeveloper will assign a unique name to identify the LOV usage. For example, the metadata for the attribute `SupplierDesc` will specify the name `SupplierDescLOV` to indicate that the attribute is LOV enabled.

4. In the **List Data Source** section, click the **Create View Accessor** icon to add the accessor to the view object you are currently editing.

The display area of this section displays all view accessors that were added to the view object you are editing.

5. In the View Accessors dialog, select the view object definition or shared view instance that defines the data source for the attribute and shuttle it to the view accessors list.

By default, the view accessor you create will display the same name as the view object. You can edit the accessor name to supply a unique name. For example, assign the name `SuppliersViewAccessor` for the `SuppliersView` view object.

The view instance is a view object usage that you have defined in the data model of a shared application module. For more information about using shared view instances in an LOV, see [Section 10.4.4, "How to Create an LOV Based on a Lookup Table"](#).

6. Click **OK** to save the view accessor definition for the view object.
7. In the **List of Values** section, expand **List Data Source** and select the view accessor you created for the base view object to use as its own data source. Then select the same attribute from this view accessor that will provide the list data for the LOV attribute.

The editor creates a default mapping between the list data source attribute and the LOV-enabled attribute. For example, the attribute SuppliersDesc from the PurchaseOrdersView view object would map to the attribute SuppliersDesc from the SuppliersViewAccessor view accessor.

The editor does not allow you to remove the default attribute mapping for the attribute for which the list is defined.

8. Optionally, when you want to specify supplemental values that your list returns to the base view object, click the **Create Return Attribute Map** icon in the **List Return Values** section and map the desired base view object attributes with attributes accessed by the view accessor.

Supplemental attribute return values are useful when you do not require the user to make a list selection for the attributes, yet you want those values, as determined by the current row, to participate in the update. For example, to map the attribute SupplierAddress from the PurchaseOrdersView view object, you would choose the attribute SupplierAddress from the SuppliersViewAccessor view accessor.

9. Click **OK**.

5.11.2 How to Define Cascading Lists for LOV-Enabled View Object Attributes

When the application user interface requires a list of values in one input field to be dependent on the user's entry in another field, you can create attributes that will display as cascading lists in the user interface. In this case, the list of possible values for the LOV-enabled attributes might be different for each row. As the user changes the current row, the LOV values vary based on the value of one or more controlling attribute values in the LOV-enabled attribute's view row. To apply the controlling attribute to the LOV-enabled attribute, you will create a view accessor to access the data source view object with the additional requirement that the accessor filters the list of possible values based on the current value of the controlling attribute. To filter the LOV-enabled attribute, you can edit the view accessor to add a named view criteria with a bind variable to obtain the user's selection.

For example, assume that a purchase order form contains a field that requires the user to select the supplier's specific site and that the available sites will depend on the order's already specified supplier. To implement this requirement, you would first create a view accessor that points to the data source view object. The data source view object will be specific to the LOV usage, because it must perform a query that filters the available supplier sites based on the user's supplier selection. You might name this data source view object definition `SupplierIdsForCurrentSupplierSite` to help distinguish it from the `SupplierSitesView` view object that the data model already contains. The data source view object will use a named view criteria (`SupplierCriteria`) with a single view criteria item set by a bind variable (`TheSupplierId`) to obtain the user's selection for the controlling attribute (`SupplierId`).

You would then set the LOV on the `SupplierSiteId` attribute of the base view object (`PurchaseOrdersView`). You can then reference the view accessor that points to the data source view object from the LOV-enabled attribute (`PurchaseOrdersView.SupplierSiteId`) of the base view object. Finally, you must edit the LOV-enabled attribute's view accessor definition to specify the corresponding attribute (`SupplierIdsForCurrentSupplierSite.SupplierSiteId`) from the view object as the data source and, importantly, source the value of the bind variable from the view row's result using the attribute `SupplierId`.

The two procedures that follow describe how to:

1. Create the data source view object and define a named view criteria to filter the data source attribute based on the value of another attribute.
2. Create a view accessor that references that named view criteria and populate the LOV-enabled attribute on the base view object.

To create a data source view object and view criteria with bind variable to filter the view accessor for a cascading LOV:

1. In the Application Navigator, double-click the view object that you created to query the list of all possible values for the controlling attribute.

For example, if the LOV-enabled attribute SupplierSiteId depends on the controlling attribute SupplierId value, you might have created the data source view object SupplierIdsForCurrentSupplierSite to query the list of all supplier sites.

2. In the overview editor navigation list, select **Query**.
3. In the **Bind Variables** section, click the **Create New Bind Variable** icon to add a bind variable to data source view object.

For example, for a data source view object SupplierIdsForCurrentSupplierSite used to query the list of all supplier sites, you would create the bind variable TheSupplierId, since it will be the controlling attribute for the LOV-enabled attribute.

4. In the Bind Variable dialog, enter the name and type of the bind variable. Leave all other options unchanged and click **OK**.

By default, the view accessor you create will display the same name as the view object instance. You can edit the accessor name to supply a unique name. For example, assign the name CurrencyLookupViewAccessor for the CurrencyLookupView view object instance.

5. In the **View Criteria** section of the overview editor, click the **Create New View Criteria** icon to add the view criteria to the data source view object you are currently editing.
6. In the Edit View Criteria dialog, click **Add Group** and define a single **Criteria Item** for the group as follows:

- Enter a **Criteria Name** to identify the view criteria. For example, you might enter the name SupplierCriteria for the SupplierIdsForCurrentSupplierSite.
- Select the controlling attribute from the **Attributes** list. For example, you would select the SupplierSiteId attribute from the SupplierIdsForCurrentSupplierSite.
- Select **equal to** from the view criteria **Operator** list.
- Select **Bind Variable** from the view criteria **Operand** list.
- Select the name of the previously defined bind variable from the **Parameter** list.
- Select **Required** from the **Usage** menu.

The WHERE clause shown in the Edit View Criteria dialog should look similar to `((SupplierIdsForCurrentSupplierSite.SUPPLIER_ID = :TheSupplierId)).`

7. Click OK.

After you have created the named view criteria with bind variable to filter the controlling attribute, you will define a view accessor on the base view object to populate the LOV-enabled attribute. Use will add the view accessor to the LOV-enabled attribute of the base view object and reference the previously defined data source view object's named view criteria.

To create a view accessor that filters display values for an LOV-enabled attribute based on the value of another attribute in the same view row:

1. In the Application Navigator, double-click the base view object that contains the attribute you want to use the filtered view accessor as the list data source.

For example, the base view object PurchaseOrdersView might contain the attribute SupplierSiteId that will depend on the value of the controlling attribute SupplierId.

2. In the Edit Attribute dialog, select **List of Values** from the navigation list and select **Enable List of Values**.
3. In the **List Data Source** section, click the **Create View Accessor** icon to add the accessor to the view object you are currently editing.

The display area of this section displays all view accessors that were added to the view object you are editing.

4. In the View Accessors dialog, select the view object instance name you created for data source view object and shuttle it to the view accessors list.
5. With the new view accessor selected in the dialog, click **Edit**.
6. In the Edit View Accessor dialog, apply the previously defined view criteria to the view accessor and provide a value for the bind variable as follows:
 - Click the data source view object's view criteria in the **Available** list and add it to the Selected list. For example, you would select SupplierCriteria from the SupplierIdsForCurrentSupplierSite view object definition.
 - Select **Row-level Bind Values Exist** in the **Bind Parameters Values** section.

Be sure that you enable **Row-level Bind Values Exist** before exiting the dialog. This selection ensures that ADF Business Components will populate the row's controlling attribute from the view accessor WHERE clause and not from the query on the base view object.

- Set the value for the bind variable to the name of the controlling attribute. The attribute name must be identical to the base view object's controlling attribute. For example, if the base view object PurchaseOrdersView contains the LOV-enabled attribute SupplierSiteId that depends on the value of the controlling attribute SupplierId, you would enter SupplierId for the bind variable value.
 - Select the name of the previously defined bind variable from the **Parameter** list.
 - Select **Required** from the **Usage** dropdown list.
7. Click **OK** to save the view accessor definition for the base view object.
 8. In the overview editor navigation list, select **Attributes** and double-click the attribute that is to display the LOV.

9. In the Edit Attribute dialog, select **List of Values** in the navigation list and select **Enable List of Values**.
10. Expand **List Data Source** and select the view accessor you created for the data source view object instance to use as the data source. Then select the controlling attribute from this view accessor that will serve to filter the attribute you are currently editing.

The editor creates a default mapping between the view object attribute and the LOV-enabled attribute. You use separate attributes in order to allow the bind variable (set by the user's controlling attribute selection) to filter the LOV-enabled attribute. For example, the LOV-enabled attribute `SupplierId` from the `PurchaseOrdersView` view object would map to the controlling attribute `SupplierSiteId` on which you would have enabled row-level bind values for the `SupplierIdsForCurrentSupplierSiteViewAccessor`. The runtime automatically supports these two cascading LOVs where the row set and the base row attribute differ.

11. Click **OK**.

5.11.3 How to Set User Interface Hints on a View Object LOV-Enabled Attribute

When you know how the view object attribute that you define as an LOV should appear in the user interface, you can specify additional properties of the LOV to determine its display characteristics. These properties, or *UI hints*, augment the attribute hint properties that ADF Business Components lets you set on any view object attribute. The LOV UI hints include the following.

- **Default List Type:** Select the type of component the user interface will use to display the list. For a description of the available components, see [Table 5–1](#). (Not all ADF Faces components support the default list types, as noted in the [Table 5–1](#).)
The web page that displays the list and the view object's default list type must match at runtime or a method-not-found runtime exception results. To avoid this error, confirm the desired list component with the user interface designer. You can also edit the default list type to match, so that, should the user interface designer subsequently change the component used in the web page, the two stay in sync.
- **Display Attributes:** Specify which additional attributes from the LOV-enabled attribute's view row, should appear in the list. The additional attribute values can help the end user select an item from the list.
- **List Search:** Provides model layer support for setting properties of the ADF Faces query component specifically for use with those list type components that display the List of Values dialog. These components are the combobox with LOV component or the input text field with LOV component. By default, the List of Values dialog will display a search form that will allow the user to search on all queryable attributes of the data source view object (the one defined by the LOV-enabled attribute's view accessor). You can seed the search form in advance by selecting a view criteria from the data source view object. Thus, by defining a search form based on a view criteria, you can effectively limit the search criteria displayed in by the List of Values dialog's search form.
- **Most Recently Used Count:** Optionally, enter the number of items to display in the choice list when you want to provide a shortcut for users to display their most recent selections. For example, your form might display a choice list of `SupplierId` values to drive a purchase order form. In this case, you can allow the user to select from a list of their most recently viewed suppliers, where the number

of supplier choices is determined by the count you enter. The default count 0 (zero) for the choice list displays all values for the attribute.

- **Include "No Selection" Item:** Select when you want the end user to be able to return a NULL value. You can also determine how the NULL value selection should appear in the list.

Table 5–1 List Component Types For List Type Control Hint

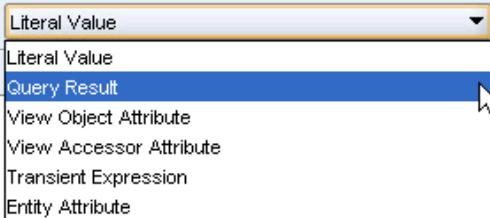
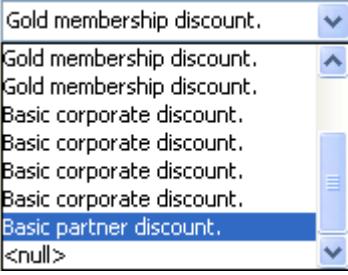
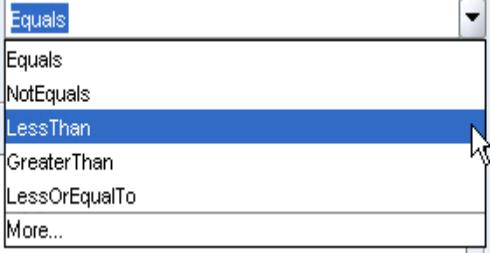
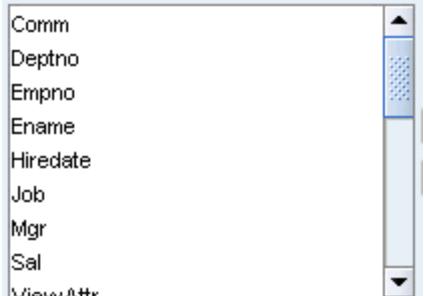
LOV List Component Type	Usage
Choice List	The user cannot type in text, only select from the dropdown list. 
Combo Box	The user can type text or select from the dropdown list. This component sometimes supports auto-complete as the user types. This component is <i>not</i> supported for ADF Faces. 
Combo Box with List of Values	Same as Combo Box, except the last entry (More...) opens a List of Values dialog that supports query with filtering when enabled for the LOV attribute in its UI hints. The default UI hint enables queries on all attributes. This component is <i>not</i> supported for ADF Faces. Note that when the LOV attribute appears in a table component, the list type changes to an Input Text with List of Values as shown below. 
Input Text with List of Values	This component displays an input text field with an LOV button next to it. The List of Values dialog opens when the user clicks the button or enters an invalid value into the text field. The List of Values dialog for this component supports query with filtering when enabled in the UI hints for the LOV attribute. The default UI hint enables queries on all attributes. This component may also support auto-complete when a unique match exists. 

Table 5–1 (Cont.) List Component Types For List Type Control Hint

LOV List Component Type	Usage
List Box	This component takes up a fixed amount of real estate on the screen and is scrollable (as opposed to the choice list, which takes up a single line until the user clicks on it).
	
Radio Group	This component displays a radio button group with the selection choices determined by the LOV attribute values. This component is most useful for very short, fixed lists.
	

Select when you want the end user to be able to return a NULL value. You can also determine how the NULL value selection should appear in the list.

To set view object attribute UI hints for an LOV-enabled attribute:

1. In the Application Navigator, double-click the view object that contains the attribute that you want to customize.
2. In the overview editor navigation list, select **Attributes** and double-click the attribute that displays the LOV.
3. In the Edit Attribute dialog, select **List of Values** in the navigation list and select the **UI Hints** tab.
4. Select a default list type as the type of component to display the list.
5. Optionally, select additional display attributes to add values to the display.
6. If you selected a list type that allows the user to open a List of Values dialog to select a list value (this includes either the Combobox with List of Values type component or Input Text with List of Values type component), by default, the dialog will display a search region to support user queries. You can customize the search region:
 - a. You can limit the attributes to display in the List of Values dialog search form by selecting a view criteria from the **Include Search Region** dropdown list. To appear in the dropdown list, the view criteria must already have been defined on the data source view object (the one that the LOV-enabled attribute's view accessor defines). Click the **Edit View Criteria** icon to set search form properties for the selected view criteria. For more information about customizing view criteria for search forms, see [Section 5.10.2, "How to Set User Interface Hints on View Criteria"](#).
 - b. You can prepopulate the results table of the List of Values dialog by selecting **Query List Automatically**. The List of Values dialog will display the results of

the query when the user opens the dialog. If you leave this option deselected, no results will be displayed until the submits the search form.

7. Alternatively, if you prefer not to display a search region in the List of Values dialog, select **<No Search>** from the **Include Search Region** dropdown list. In this case, the List of Values dialog will display only attributes you add to the **Display Attributes** list.
8. If you selected a choice type component to display the list, you can optionally specify a **Most Recently Used Count** as an alternative to displaying all possible values. You can also decide how you want the list component to handle the case where you enable **Include No Selection Item**.
9. Click **OK** to save the changes and return to the Edit Attribute dialog.
10. Click **OK**.

5.11.4 How to Test LOV-Enabled Attributes Using the Business Component Browser

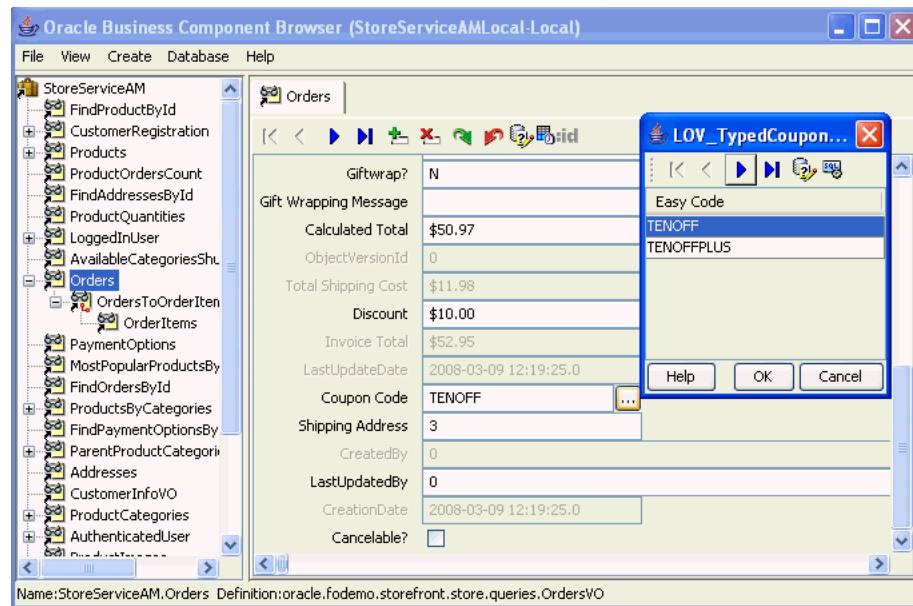
To test the LOV you created for a view object attribute, use the Business Component Browser, which is accessible from the Application Navigator.

The Business Component Browser, for any view object instance that you browse, will display any LOV-enabled attributes using one of two component types you can select in the UI Hints page of the List of Values dialog. Currently, only a Choice List component type and Input Text with List of Values component type are supported. Otherwise, the Business Component Browser uses the default choice list type to display the LOV-enabled attribute.

To test an LOV using the Business Component Browser:

1. In the Application Navigator, expand the project containing the desired application module and view objects.
2. Right-click the application module and choose **Run**.
3. In the Select Business Components Configuration dialog, select the desired application module configuration from the **Configuration Name** list to run the Business Component Browser.
4. Click **Connect** to start the application module using the selected configuration.
5. In the Business Component Browser, select the desired view object from the section on the left. The Business Component Browser displays the LOV-enabled attribute as a dropdown choice list unless you specified the component type as an Input Text with List of Value, UI hint.

[Figure 5–39](#) shows an LOV-enabled attribute, `TypeCouponCode` for the `OrdersVO`, that specifies an input text field and List of Values dialog as the UI hint list type. The Input Text with List of Values component is useful when you want to display the choices in a separate LOV dialog. Other list types are not supported by the Business Component Browser.

Figure 5–39 Displaying LOV-Enabled Attributes in the Business Component Browser

5.11.5 What Happens When You Define an LOV for a View Object Attribute

When you define an LOV for a view object attribute, the view object metadata defines the following additional information, as shown in [Example 5–34](#) for the OrdersVO.TypedCouponCode attribute in the Fusion Order Demo application.

- The <ViewAttribute> element names the attribute, points to the list binding element that defines the LOV behavior, and specifies the component type to display in the web page. For example, the LOV-enabled attribute TypedCouponCode points to the list binding named LOV_TypedCouponCode and defines the CONTROLTYPE input text field with list (input_text_lov) to display the LOV data.

When the user interface designer creates the web page using the Data Controls panel, the <CONTROLTYPE Value="namedType" /> definition determines the component that JDeveloper will add to the web page. When the component type definition in the data model project does not match the component type displayed in the web page, a runtime exception will result. For more information, see [Section 5.11.6, "What Happens at Runtime: When an LOV Queries the List Data Source"](#).

- The <ListBinding> element defines the behavior of the LOV. It also identifies a view accessor to access the data source for the LOV-enabled attribute. The view accessor is the ADF Business Components mechanism that lets you obtain the full list of possible values from the row set of the data source view object. For example, ListVOName="Coupon" points to the Coupons view accessor, which accesses the view object CouponsVO.
- The <ListBinding> element maps the list data source attribute to the LOV-enabled attribute. For example, the ListAttrNames item EasyCode is mapped to the LOV-enabled attribute TypedCouponCode.
- Optionally, the <ListBinding> element defines supplemental values that the data source may return to attributes of the base view object other than the data source attribute for which the list is defined. For example, DerivedAttrNames

- item CouponId is a supplemental value set by the ListAttrNames item DiscountId.
- The <ListBinding> element also identifies one or more attributes to display from the current row and provides a few options that are specific to the choice list type component. For example, the ListDisplayAttrNames item EasyCode is the only attribute displayed by the LOV-enabled attribute TypedCouponCode. In this example, the value none for NullValueFlag means the user cannot select a blank item from the list.

Example 5–34 View Object MetaData For LOV-Attribute Usage

```

<ViewAttribute
    Name="TypedCouponCode"
    LOVName="LOV_TypedCouponCode"
    . .
    <Properties>
        <SchemaBasedProperties>
            <CONTROLYTYPE Value="input_text_lov" />
        </SchemaBasedProperties>
    </Properties>
</ViewAttribute>
. .
<ListBinding
    Name="LOV_TypedCouponCode"
    ListVOName="Coupons"
    ListRangeSize="-1"
    NullValueFlag="none"
    NullValueId="LOV_TypedCouponCode_NullValueId"
    MRUCount="0">
    <AttrArray Name="AttrNames">
        <Item Value="TypedCouponCode" />
    </AttrArray>
    <AttrArray Name="DerivedAttrNames">
        <Item Value="CouponId" />
    </AttrArray>
    <AttrArray Name="ListAttrNames">
        <Item Value="EasyCode" />
        <Item Value="DiscountId" />
    </AttrArray>
    <AttrArray Name="ListDisplayAttrNames">
        <Item Value="EasyCode" />
    </AttrArray>
</ListBinding>
. .
<ViewAccessor
    Name="Coupons"
    ViewObjectName="oracle.fodemo.storefront.store.queries.CouponsVO" />
```

5.11.6 What Happens at Runtime: When an LOV Queries the List Data Source

The ADF Business Components runtime adds view accessors in the attribute setters of the view row and entity object to facilitate the LOV-enabled attribute behavior. In order to display the LOV-enabled attribute values in the user interface, the LOV facility fetches the data source, and finds the relevant row attributes and mapped target attributes.

The number of data objects that the LOV facility fetches is determined in part by the ListRangeSize setting in the LOV-enabled attribute's list binding definition. By

default, the `ListRangeSize` item is set to `-1`. This means the user will be able to view all the data objects from the data source. The default value places no restrictions on the number of records the list component can display. Although you can alter the `ListRangeSize` value in the metadata definition for the `<ListBinding>` element, setting the value to a discrete number of records (for example, `ListRangeSize="10"`) most likely will not provide the user with the desired selection choices.

Performance Tip: To limit the set of values a LOV displays use a view accessor to filter the LOV binding, as described in [Section 5.11.1, "How to Define a Single LOV-Enabled View Object Attribute"](#).

Additionally, in the case of component types that display a choice list, you can change the **Most Recently Used Count** setting to limit the list to display the user's previous selections, as described in [Section 5.11.3, "How to Set User Interface Hints on a View Object LOV-Enabled Attribute"](#).

Note, a runtime exception will occur when a web page displays a UI component for an LOV-enabled attribute that does not match the view object's `CONTROLYTYPE` definition. When the user interface designer creates the page in JDeveloper using the Data Controls panel, JDeveloper automatically inserts the list component identified by the **Default List Type** selection you made for the view object's LOV-enabled attribute in the List UI Hint dialog. However, if the user interface designer changes the list type subsequent to creating the web page, you will need to edit the selection in the List UI Hint dialog to match.

5.11.7 What You May Need to Know About Lists

There are several things you may need to know about LOVs that you define for attributes of view objects, including how to propagate LOV-enabled attributes from parent view objects to child view objects (by extending an existing view object) and when to use validators instead of an LOV to manage a list of values.

5.11.7.1 Inheritance of AttributeDef Properties from Parent View Object Attributes

When a view object extends another view object, you can create the LOV-enabled attribute on the base object. Then when you define the child view object in the overview editor, the LOV definition will be visible on the corresponding view object attribute. This inheritance mechanism allows you to define an LOV-enabled attribute once and later apply it across multiple view objects instances for the same attribute.

You can also use the overview editor to extend the inherited LOV definition. For example, you may add extra attributes already defined by the base view object's query to display in selection list. Alternatively, you can define a view object that uses a custom `WHERE` clause to query the supplemental attributes not already queried by the based view object. For information about customizing entity-based view objects, see [Section 5.9, "Working with Bind Variables"](#).

5.11.7.2 Using Validators to Validate Attribute Values

If you have created an LOV-enabled attribute for a view object, there is no need to validate the attribute using a List Validator. You only use an attribute validator when you do not want the list to display in the user interface, but still need to restrict the list of valid values. List validation may be a simple static list or it may be a list of possible values obtained through a view accessor you define. Alternatively, you might prefer to use Key Exists validation when the attribute displayed in the UI is one that references a key value (such as a primary, foreign, or alternate key). For information about

declarative validation in ADF Business Components, see [Chapter 7, "Defining Validation and Business Rules Declaratively"](#).

5.12 Defining Attribute Control Hints for View Objects

One of the built-in features of Oracle ADF Business Components is the ability to define control hints on attributes. Control hints are additional attribute settings that the view layer can use to automatically display the queried information to the user in a consistent, locale-sensitive way. JDeveloper stores the hints in resource bundle files that you can easily localize for multilingual applications.

5.12.1 How to Add Attribute Control Hints

To create control hints for attributes of a view object, use the overview editor for the view object, which is accessible from the Application Navigator. You can also display and edit control hints using the Property Inspector that you display for an attribute.

To customize view object attribute with control hints:

1. In the Application Navigator, double-click the view object for which you want to define attribute control hints.
2. In the overview editor navigation list, select **Attributes**.
3. In the **Attributes** section, double-click the attribute that you want to customize with control hints.

Alternatively, display the Property Inspector for the selected attribute and select the **UI Hints** navigation tab. The Property Inspector provides a way to customize the attribute's control hints without using the Edit Attribute dialog.

4. In the Edit Attribute dialog, select **Control Hints** and define the desired hints.
- For example, for an attribute `UserId`, you might enter a value for its **Label Text** hint like "Id" or set the **Format Type** to Number, and enter a **Format** mask of 00000.
5. Click **OK**.

Note: Java defines a standard set of format masks for numbers and dates that are different from those used by the Oracle database's SQL and PL/SQL languages. For reference, see the Javadoc for the `java.text.DecimalFormat` and `java.text.SimpleDateFormat` classes.

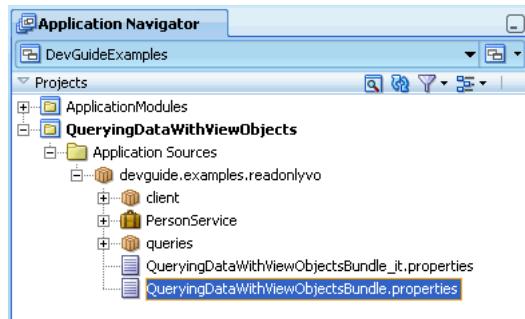
5.12.2 What Happens When You Add Attribute Control Hints

When you define attribute control hints for a view object, by default JDeveloper creates a project-level resource bundle file in which to store them. For example, when you define control hints for a view object in the `StoreFront` project, JDeveloper creates the message bundle file named `StoreFrontBundle.xxx` for the package. The hints that you define can be used by generated forms and tables in associated view clients.

The type of resource bundle file that JDeveloper uses and the granularity of the file are determined by settings on the Resource Bundle page of the Project Properties dialog. By default, JDeveloper sets the option to **Properties Bundle** and generates one `.properties` file for the entire data model project.

Alternatively, if you select the option in the Project Properties dialog to generate one resource bundle per file, you can inspect the message bundle file for any view object by selecting the object in the Application Navigator and looking in the corresponding **Sources** node in the Structure window. The Structure window shows the implementation files for the component you select in the Application Navigator. You can inspect the resource bundle file for the view object by expanding the parent package of the view object in the Application Navigator, as shown in [Figure 5–40](#).

Figure 5–40 Resource Bundle File in Application Navigator



For more information on the resource bundle options you can select, see [Section 4.7.1, "How to Set Message Bundle Options"](#).

[Example 5–35](#) shows a sample message bundle file where the control hint information appears. The first entry in each `String` array is a message key; the second entry is the locale-specific `String` value corresponding to that key.

Example 5–35 Resource File With Locale-Sensitive Control Hints

```
devguide.examples.readonlyvo.queries.Persons.PersonId_FMT_FORMATTER=
    oracle.jbo.format.DefaultNumberFormatter
devguide.examples.readonlyvo.queries.Persons.PersonId_FMT_FORMAT=00000
devguide.examples.readonlyvo.queries.Persons.PersonId_LABEL=Id
devguide.examples.readonlyvo.queries.Persons.Email_LABEL=Email Address
devguide.examples.readonlyvo.queries.Persons.LastName_LABEL=Surname
devguide.examples.readonlyvo.queries.Persons.FirstName_LABEL=Given Name
```

5.12.3 What You May Need to Know About Resource Bundles

Internationalizing the model layer of an application built using ADF Business Components entails producing translated versions of each component's resource bundle file. For example, the Italian version of the `QueryDataWithViewObjectsBundle.properties` file would be a file named `QueryDataWithViewObjectsBundle_it.properties`, and a more specific Swiss Italian version would have the name `QueryDataWithViewObjectsBundle_it_ch.properties`.

Resource bundle files contain entries for the message keys that need to be localized, together with their localized translation. For example, assuming you didn't want to translate the number format mask for the Italian locale, the Italian version of the `QueryDataWithViewObjects` view object message keys would look like what you see in [Example 5–36](#). At runtime, the resource bundles are used automatically, based on the current user's locale settings.

Example 5–36 Localized View Object Component Resource Bundle for Italian

```
devguide.examples.readonlyvo.queries.Persons.PersonId_FMT_FORMATTER=
```

```

oracle.jbo.format.DefaultNumberFormatter
devguide.examples.readonlyvo.queries.Persons.PersonId_FMT_FORMAT=00000
devguide.examples.readonlyvo.queries.Persons.PersonId_LABEL=Codice Utente
devguide.examples.readonlyvo.queries.Persons.Email_LABEL=Indirizzo Email
devguide.examples.readonlyvo.queries.Persons.LastName_LABEL=Cognome
devguide.examples.readonlyvo.queries.Persons.FirstName_LABEL=Nome

```

5.13 Adding Calculated and Transient Attributes to a View Object

In addition to having attributes that map to underlying entity objects, your view objects can include calculated attributes that don't map to any entity object attribute value. The two kinds of calculated attributes are known as:

- *SQL-calculated attributes*, when their value is retrieved as an expression in the SQL query's SELECT list
- *Transient attributes*, when their value is not retrieved as part of the query

A view object can include an entity-mapped attribute which itself is a transient attribute at the entity object level.

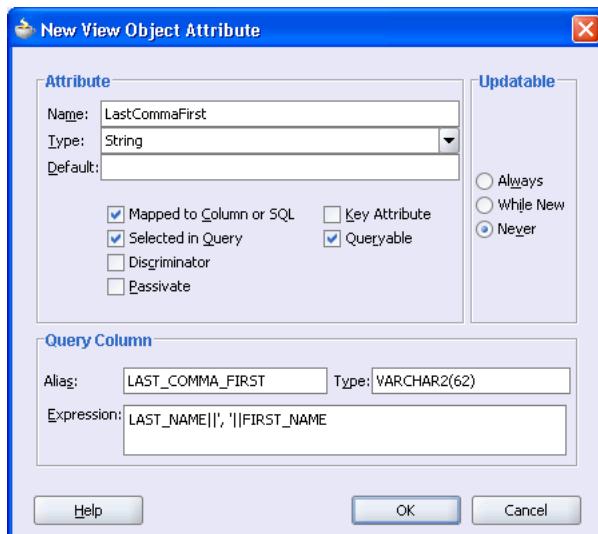
5.13.1 How to Add a SQL-Calculated Attribute

You use the overview editor for the view object to add a SQL-calculated attribute.

To add a SQL-calculated attribute to a view object:

1. In the Application Navigator, double-click the view object for which you want to define SQL-calculated attribute.
2. In the overview editor navigation list, select **Attributes**.
3. In the Attributes page, click **Create new attribute**.
4. In the New View Object Attribute dialog, enter a name for the attribute.
5. Set the Java attribute type to an appropriate value.
6. Select the **Mapped to Column or SQL** checkbox.
7. Provide a SQL expression in the **Expression** field.

For example, to change the order of first name and last name, you could write the expression LAST_NAME || ' , ' || FIRST_NAME, as shown in [Figure 5–41](#).

Figure 5–41 New SQL-Calculated Attribute

8. Consider changing the SQL column alias to match the name of the attribute.
9. Verify the database query column type and adjust the length (or precision/scale) as appropriate.
10. Click **OK**.

5.13.2 What Happens When You Add a SQL-Calculated Attribute

When you add a SQL-calculated attribute in the overview editor for the view object, JDeveloper updates the XML component definition for the view object to reflect the new attribute. The entity-mapped attribute's `<ViewAttribute>` tag looks like the sample shown in [Example 5–37](#). The entity-mapped attribute inherits most of its properties from the underlying entity attribute to which it is mapped.

Example 5–37 Metadata For Entity-Mapped Attribute

```
<ViewAttribute
    Name="LastName"
    IsNotNull="true"
    EntityAttrName="LastName"
    EntityUsage="User1"
    AliasName="LAST_NAME" >
</ViewAttribute>
```

Whereas, in contrast, a SQL-calculated attribute's `<ViewAttribute>` tag looks like the sample shown in [Example 5–38](#). As expected, the tag has no `EntityUsage` or `EntityAttrName` property, and includes datatype information along with the SQL expression.

Example 5–38 Metadata For SQL-Calculated Attribute

```
<ViewAttribute
    Name="LastCommaFirst"
    IsUpdatable="false"
    IsPersistent="false"
    Precision="62"
    Type="java.lang.String"
    ColumnType="VARCHAR2"
```

```

AliasName="FULL_NAME"
Expression="LAST_NAME||' ', |FIRST_NAME"
SQLType="VARCHAR" >
</ViewAttribute>

```

Note: The ' is the XML character reference for the apostrophe. You reference it by its numerical ASCII code of 39 (decimal). Other characters in literal text that require similar construction in XML are the less-than, greater-than, and ampersand characters.

5.13.3 How to Add a Transient Attribute

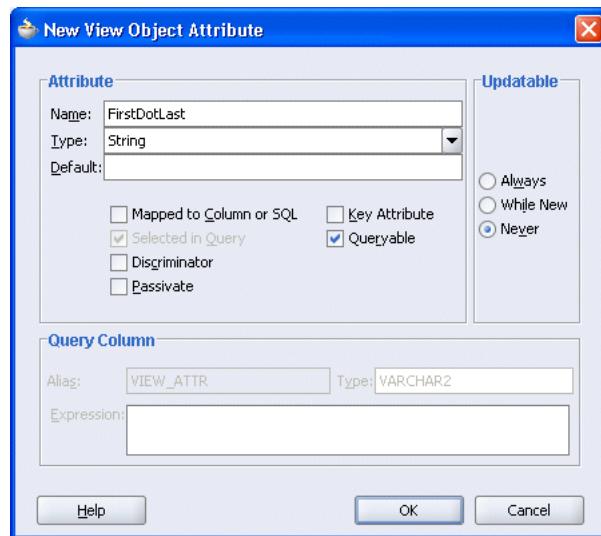
Transient attributes are often used to provide subtotals or other calculated expressions that are not stored in the database.

To add a transient attribute to a view object:

1. In the Application Navigator, double-click the view object for which you want to define transient attribute.
2. In the overview editor navigation list, select **Attributes**.
3. In the Attributes page, click **Create new attribute**.
4. In the New View Object Attribute dialog, enter a name for the attribute.
5. Set the Java attribute type to an appropriate value.

For example, a calculated attribute that concatenates a first name and a last name would have the type **String**, as shown in [Figure 5–42](#).

Figure 5–42 New Transient Attribute



6. Leave the **Mapped to Column or SQL** checkbox unselected.
7. Click **OK**.

To create a transient attribute based on an expression:

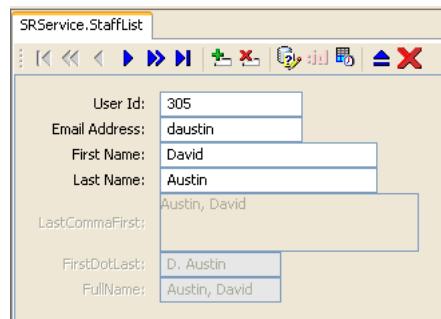
1. In the Application Navigator, double-click the view object for which you want to define SQL-calculated attribute.

2. In the overview editor navigation list, select **Attributes**.
 3. In the Attributes page, click **Create new attribute**.
 4. In the New View Object Attribute dialog, enter a name for the attribute.
 5. Set the Java attribute type to an appropriate value.
 6. Leave the **Mapped to Column or SQL** checkbox unselected.
A transient attribute does not include a SQL expression.
 7. Next to the **Value** field, click **Edit** to define an expression that calculates the value of the attribute.
Expressions you define will be evaluated using the Groovy Expression Language. Groovy lets you insert expressions and variables into strings. The expression will be saved as part of the view object definition. For more information about Groovy, see [Section 3.6, "Overview of Groovy Support"](#).
 8. In the Edit Expression dialog, enter an expression in the field provided.
Attributes that you reference can include any attribute that the base entity objects define. Do not reference attributes in the expression that are not defined by the view object's underlying entity objects.
 9. 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.
 10. You can optionally provide a condition for when to recalculate the expression.
For example, the following expression in the **Based on the following expression** field causes the attribute to be recalculated when either the **Quantity** attribute or the **UnitPrice** attribute are changed:


```
return (adf.object.isAttributeChanged("Quantity") ||
adf.object.isAttributeChanged("UnitPrice"));
```
 11. When either the value expression or the optional recalculate expression that you define references an attribute from the base entity object, you must define this as a dependency in the Edit Expression dialog. Locate each attribute in the **Available** list and shuttle it to the **Selected** list.
 12. Click **OK** to save the expression and return to the New View Object Attribute dialog.
 13. Click **OK**.
- A view object can include an entity-mapped attribute which itself is a transient attribute at the entity object level.
- To add a transient attribute from an entity object to an entity-based view object:**
1. In the Application Navigator, double-click the view object for which you want to add a transient attribute based on an entity usage.
 2. In the overview editor navigation list, select **Attributes**.
 3. In the Attributes page, click **Add from Entity**.
 4. In the Attributes dialog, move the desired transient attribute from the **Available** list into the **Selected** list.
 5. Click **OK**.

If you use the Business Component Browser to test the data model, you can see the usage of your transient attributes. [Figure 5–43](#) shows three attributes that were created using a SQL-calculated attribute (`LastCommaFirst`), a transient attribute (`FirstDotLast`) and an entity-derived transient attribute (`FullName`).

Figure 5–43 View Object with Three Kinds of Calculated Attributes



5.13.4 What Happens When You Add a Transient Attribute

When you add a transient attribute in the overview editor for a view object, JDeveloper updates the XML component definition for the view object to reflect the new attribute. A transient attribute's `<ViewAttribute>` tag in the XML is similar to the SQL-calculated one, but it lacks an `Expression` property.

When you base a transient attribute on a Groovy expression, a `<TransientExpression>` tag is created within the appropriate attribute, as shown in [Example 5–39](#).

Example 5–39 Calculating a Transient Attribute Using a Groovy Expression

```
<TransientExpression>
  <! [CDATA[
    ((Quantity == null) ? 0 : Quantity) * ((UnitPrice == null) ? 0 : UnitPrice)
  ]>
</TransientExpression>
```

5.13.5 Adding Java Code in the View Row Class to Perform Calculation

A transient attribute is a placeholder for a data value. If you change the **Updatable** property of the transient attribute to **While New** or **Always**, the end user can enter a value for the attribute. If you want the transient attribute to display a calculated value, then you'll typically leave the **Updatable** property set to **Never** and write custom Java code that calculates the value.

After adding a transient attribute to the view object, to make it a *calculated* transient attribute you need to enable a custom view row class and choose to generate accessor methods, in the Java dialog that you open clicking the **Edit** icon on the Java page of the overview editor for the view object. Then you would write Java code inside the accessor method for the transient attribute to return the calculated value.

[Example 5–39](#) shows the `StaffListRowImpl.java` view row class contains the Java code to return a calculated value in the `getLastCommaFirst()` method.

```
// In StaffListRowImpl.java
public String getFirstDotLast() {
  // Commented out this original line since we're not storing the value
  // return (String)getAttributeInternal(FIRSTDOTLAST);
  return getFirstName().substring(0,1)+"."+getLastName();
```

```
}
```

Note: [Section 34.9, "Implementing Automatic Attribute Recalculation"](#) describes the coding technique to cause calculated attributes at the entity row level to be recalculated when one of the attribute values on which they depend is modified. You could adopt a very similar strategy at the view row level to cause automatic recalculation of calculated view object attributes, too.

5.13.6 What You May Need to Know About Transient Attributes

The view object includes the SQL expression for your SQL-calculated attribute in the SELECT list of its query at runtime. The database is the one that evaluates the expression, and it returns the result as the value of that column in the query. The value is reevaluated each time you execute the query.

6

Working with View Object Query Results

This chapter describes how to interactively test view objects query results using the Business Component Browser provided in JDeveloper. This chapter also explains how to use the Business Components API to access view object instances in a test client outside of JDeveloper.

This chapter includes the following sections:

- [Section 6.1, "Introduction to View Object Runtime Behavior"](#)
- [Section 6.2, "Creating an Application Module to Test View Instances"](#)
- [Section 6.3, "Testing View Object Instances Using the Business Component Browser"](#)
- [Section 6.4, "Testing View Object Instances Programmatically"](#)

6.1 Introduction to View Object Runtime Behavior

JDeveloper includes an interactive application module testing tool that you can use to test all aspects of its data model without having to use your application user interface or write a test client program. Running the Business Component Browser can often be the quickest way of exercising the data functionality of your business service during development.

Note: When you want to test an application module programmatically, you can write a test client. For more information, see [Section 6.4.2, "How to Create a Command-Line Java Test Client"](#).

Using the Business Component Browser, 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 your UI pages constructed, you will come to appreciate using the Business Component Browser to assist in diagnosing problems when they arise. You can reproduce the issues in the Business Component Browser to discover if the issue lies in the view or controller layers of the application, or is instead a problem in the business service layer application module itself.

6.2 Creating an Application Module to Test View Instances

Before you can test view objects that you create in your data model project, you must create an application module where you will define instances of the view objects you want to test. The application module is the transactional component that the Business Component Browser (or UI client) will use to work with application data. The set of

view objects used by an application module defines its *data model*, in other words, the set of data that a client can display and manipulate through a user interface.

To create an application module, use the Create Application Module wizard, which is available in the New Gallery.

To create an application module to test individual view object instances:

1. In the Application Navigator, right-click the project in which you want to create the application module and choose **New**.
2. In the New Gallery, expand **Business Tier**, select **ADF Business Components**, and then **Application Module**, and click **OK**.
3. In the Items list, select **Application Module** to launch the Create Application Module wizard.
4. In the Create Application Module wizard, in the Name page, provide a package name and an application module name. Click **Next**.
5. On the Data Model page, include instances of the view objects you have previously defined and edit the view object instance names to be exactly what you want clients to see. Then click **Finish**.

Instead of accepting the default instance name shown in the Data Model page, you can change the instance name to something more meaningful (for example, instead of the default name `OrderItems1` you can rename it to `AllOrderItems`).

You can also use the Create Application Module wizard to create a hierarchy of view objects for an application module, based on master-detail relationships that the view objects represent. For details about creating hierarchical relationships between view objects, see [Section 5.6, "Working with Multiple Tables in a Master-Detail Hierarchy"](#)

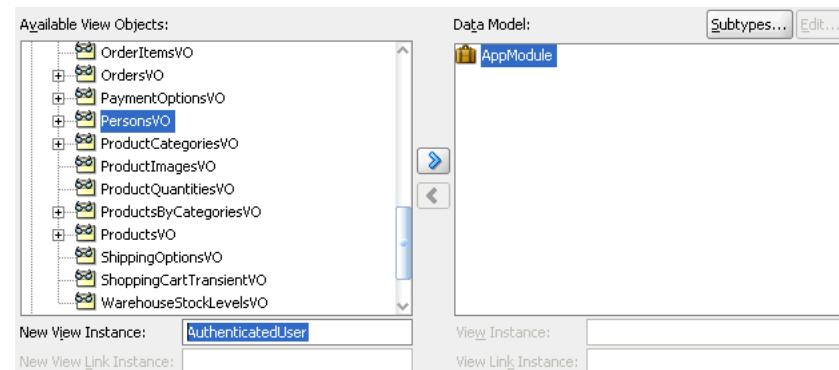
To create an application module based on view object relationships:

1. In the Create Application Module wizard, select the **Data Model** node.
2. In the **Available View Objects** list on the left, select the instance of the view object that you want to be the actively coordinating master.

The master view object will appear with a plus sign in the list indicating the available view links for this view object. The view link must exist to define a master-detail hierarchy.

For example, [Figure 6-1](#) shows `PersonsVO` selected and renamed `AuthenticatedUser` in the **New View Instance** field.

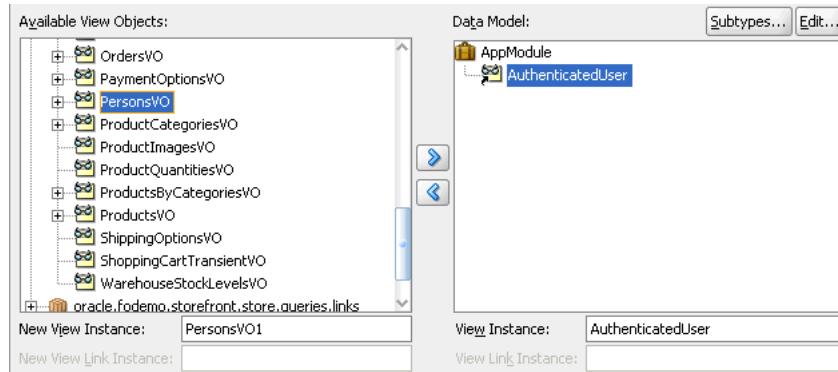
Figure 6-1 Master View Object Selected



3. Shuttle the selected master view object to the **Data Model list**

For example, [Figure 6–2](#) shows the newly created master view instance `AuthenticatedUser` in the **Data Model** list after you add it to the list.

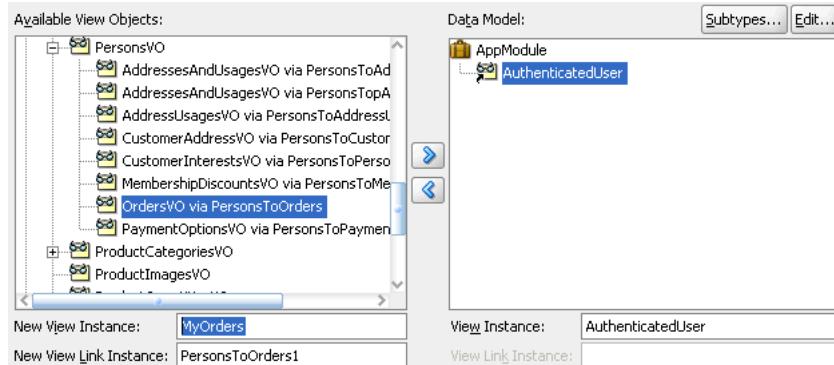
Figure 6–2 Master View Instance Created



- 4. In the **Data Model** list, leave the newly created master view instance selected, so that it appears highlighted. This will be the target of the detail view instance you will add. Then locate and select the detail view object beneath the master view object in the **Available View Objects** list.**

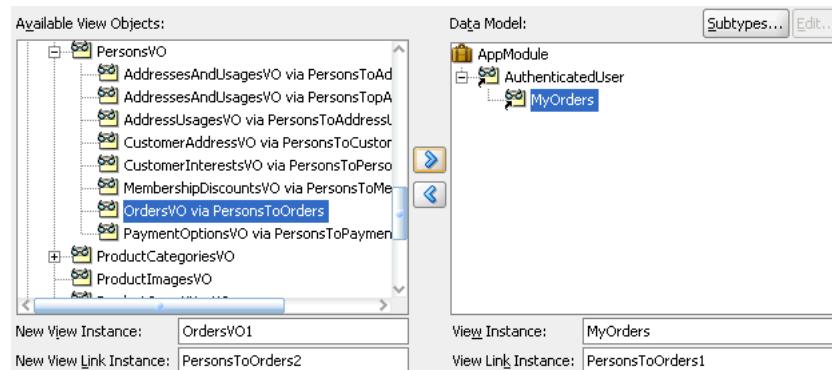
For example, [Figure 6–3](#) shows the detail `OrdersVO` indented beneath master `PersonsVO` with the name `OrdersVO` via `PersonsToOrders`. The name identifies the view link `PersonsToOrders`, which defines the master-detail hierarchy between `PersonsVO` and `OrdersVO`. The detail view instance is renamed to `MyOrders`.

Figure 6–3 Detail View Object Selected



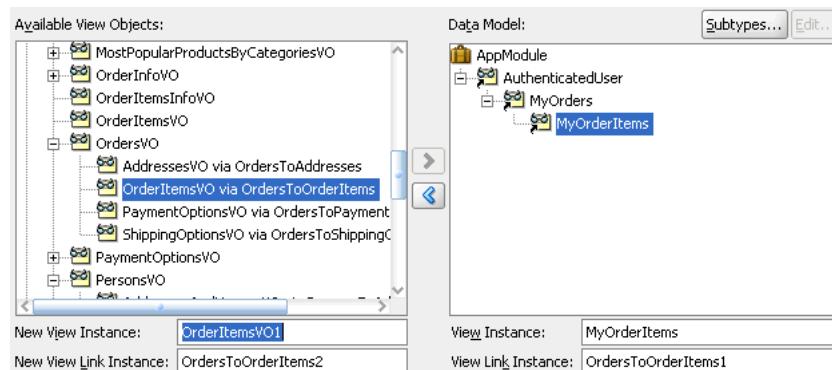
- 5. To add the detail instance to the previously added master instance, shuttle the detail view object to the **Data Model** list below the selected master view instance.**

[Figure 6–4](#) shows the newly created detail view instance `MyOrders` is a detail of the `AuthenticatedUser` in the data model.

Figure 6–4 Master View Instance Created

- To add another level of hierarchy, select the newly added detail in the **Data Model** list, then shuttle over the new detail which itself has a master-detail relationship with the previously added detail instance.

For example, [Figure 6–5](#) shows the **Data Model** list with instance `AuthenticatedUser` (renamed for `PersonsVO`) as the master of `MyOrders` (renamed for `OrdersVO` via `PersonsToOrders`), which in turn is a master for `MyOrderItems` (renamed from `OrderItemsVO` via `OrdersToOrderItems`).

Figure 6–5 Master-Detail-Detail Hierarchy Created

To test the view objects you added to an application module, use the Business Component Browser, which is accessible from the Application Navigator.

6.3 Testing View Object Instances Using the Business Component Browser

Using the Business Component Brower, 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 your UI pages constructed, you will come to appreciate using the Business Component Brower to assist in diagnosing problems when they arise. You can reproduce the issues in the Business Component Brower to discover whether the problem lies in the view or controller layers of the application, or whether there is instead a problem in the business service layer application module itself.

6.3.1 How to Run the Business Component Browser

To test the view objects you added to an application module, use the Business Component Browser, which is accessible from the Application Navigator.

To test view objects in an application module configuration:

1. In the Application Navigator, expand the project containing the desired application module and view objects.
2. Right-click the application module and choose **Run**.

Alternatively, choose **Debug** when you want to run the application in the Business Component Browser with debugging enabled. JDeveloper opens the debugger process panel in the Log window and the various debugger windows. For example, when debugging using the Business Component Browser, you can view status message and exceptions, step in and out of source code, and manage breakpoints.

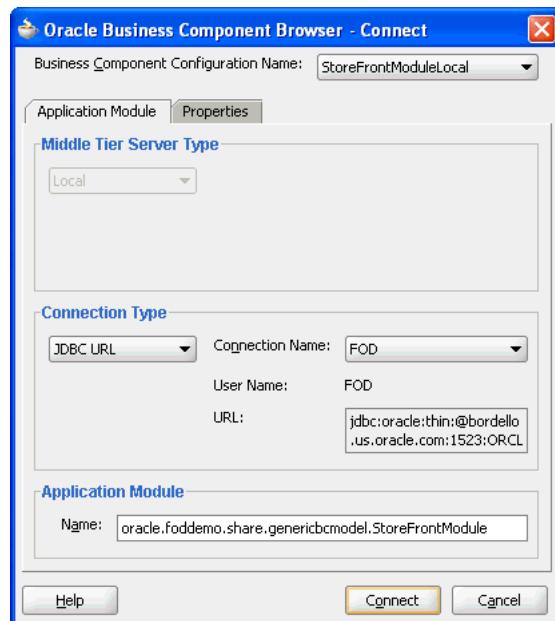
For information about receiving diagnostic messages specific to ADF Business Component debugging, see [Section 6.3.7, "How to Enable ADF Business Components Debug Diagnostics"](#).

3. In the Select Business Components Configuration dialog, choose the desired application module configuration from the **Business Component Configuration Name** list to run the Business Component Browser.

By default, an application module has only its default configurations, named *AppModuleNameLocal* and *AppModuleNameShared*. For example, [Figure 6–6](#) shows the *StoreFrontModuleLocal* configuration used by the application module to connect to the database.

If you have created additional configurations for your application module and want to test it using one of those instead, just select the desired configuration from the **Business Components Configuration** dropdown list on the Configuration dialog before clicking **Connect**.

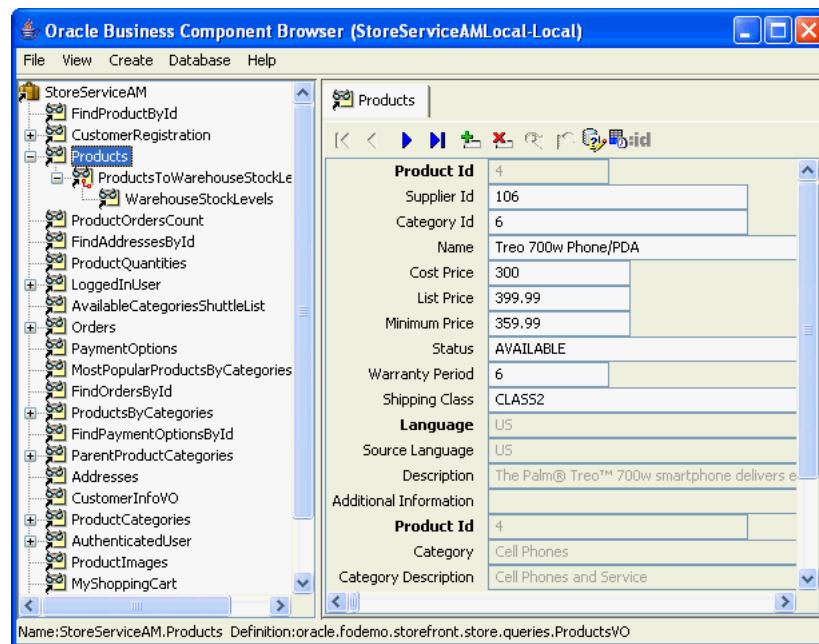
Figure 6–6 Configuration Selection in Configuration Dialog



4. Click **Connect** to start the application module using the selected configuration.
5. To execute a view object in the Business Component Browser, expand the data model tree and double-click the desired view object node.

Note that the view object instance may already appear executed in the testing session. In this case, the Business Component Browser data view page on the right already displays query results for the view object instance. The fields in the Business Component Browser data view page of a read-only view object will always appear disabled since the data it represents is not editable. For example, in [Figure 6–7](#), data for the view instance **Products** appears in the Browser. Fields like **Product Id**, **Language**, and **Category** appear disabled because the attributes themselves are not editable.

Figure 6–7 Testing the Data Model in the Business Component Browser



6. Right-click a node in the data model tree at the left of the Business Component Browser to display the context menu for that node. For example, on a view object node you can reexecute the query if needed, to remove the view object from the data model tree, and perform other tasks.
7. Right-click the tab of an open data viewer to display the context menu for that tab, as shown in [Figure 6–8](#). For example, you can close the data viewer or open it in a separate window.

Figure 6–8 Context Menu for Data Viewer Tabs in the Business Component Browser



6.3.2 How to Test Entity-Based View Objects Interactively

You test entity-based view objects interactively in the same way as read-only ones. Just add instances of the desired view objects to the data model of some application

module, and then test that application module using the Business Component Browser.

You'll find the Business Component Browser tool invaluable in quickly testing and debugging your application modules. [Figure 6–9](#) gives an overview of the operations that all of the Business Component Browser toolbar buttons perform when you display an entity-based view object.

Figure 6–9 Business Component Browser Functionality for Updatable Data Models



To test the entity-based view objects you added to an application module, use the Business Component Browser, which is accessible from the Application Navigator.

To test entity-based view objects using an application module configuration:

1. Select the application module in the Application Navigator and choose **Run** from the context menu.
2. Click **Connect** on the Select Business Component Browser Configuration dialog and use the desired configuration for testing.
3. To execute an entity-based view object in the Business Component Browser, expand the data model tree and double-click the desired view object node.

Unlike the fields of a read-only view object, the fields displayed in the data view page will appear enabled, because the data it represents is editable.

4. You can use the editable fields to update individual values and perform validation checks on the entered data.

In the case of a view instance with referenced entities, you can change the foreign key value and observe that the referenced part changes.

5. You can use the toolbar buttons to perform row-level operations, such as navigate rows, create row, remove row, and validate the current row.

For further discussion about simulating end-user interaction in the data view page, see [Section 6.3.4, "How to Simulate End-User Interaction in the Business Component Browser"](#).

6.3.3 What Happens When You Use the Business Component Browser

When you launch the Business Component Browser, JDeveloper starts the tool in a separate process and the Business Component Browser appears. The tree at the left of the dialog displays all of the view object instances in your application module's data model. After you double-click the desired view object instance, the Business Component Browser will display a data view page to inspect the query results. For

example, [Figure 6–7](#) shows the view instance `Products` that has been double-clicked in the expanded tree to display the data for this view instance in the data view page on the right.

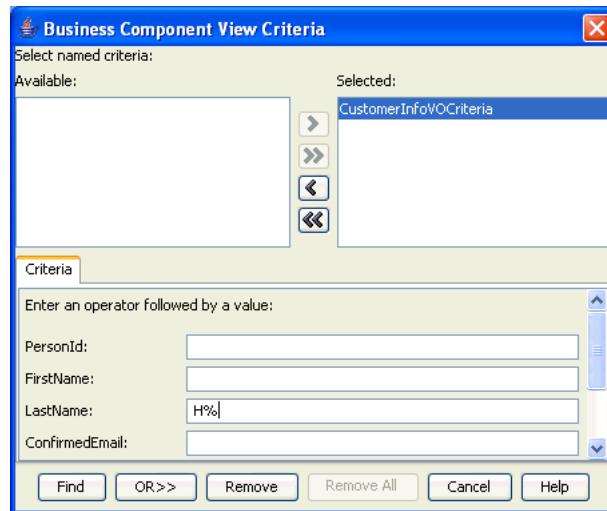
The data view page will appear disabled for any read-only view objects you display because the data is not editable. But even for a read-only view object, the tool affords some useful features:

- You can validate that the UI hints based on the Label Text control hint and format masks are defined correctly.
- You can also scroll through the data using the toolbar buttons.
- You can enter Query-by-Example criteria to find a particular row whose data you want to inspect. By clicking the **Specify View Criteria** button in the toolbar, the View Criteria dialog displays the list of available Query-by-Example criteria.

For example, as shown in [Figure 6–10](#), you can select a view criteria like `CustomerInfoVOCriteria` and enter a query criteria like "H%" for a `LastName` attribute and click **Find** to narrow the search to only those users with a last name that begins with the letter H.

The Business Component Browser becomes even more useful when you create entity-based view objects that allow you to simulate inserting, updating, and deleting rows, as described in [Section 6.3.2, "How to Test Entity-Based View Objects Interactively"](#).

Figure 6–10 Built-in Query-by-Example Functionality

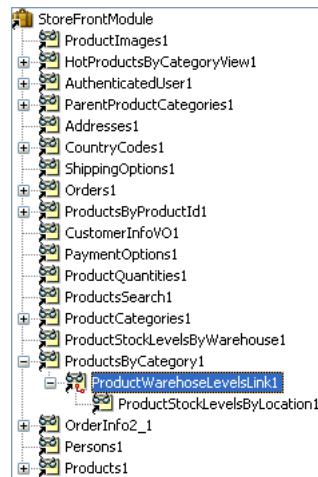


6.3.4 How to Simulate End-User Interaction in the Business Component Browser

When you launch the Business Component Browser, the tree at the left of the display shows the hierarchy of the view object instances that the data model of your application module defines. If the data model defines master-detail view instance relationships, the tree will display them as parent and child nodes. A node between the master-detail view instances represent the view link instance that performs the active master-detail coordination as the current row changes in the master. For example, in [Figure 6–11](#) the tree is expanded to show the master-detail relationship between the master `ProductByCategory1` view instance and the detail `ProductStockLevelsByLocation1` view instance. The selected node,

ProductWarehousesLevelsLink1, is the view link instance that defines the master-detail relationship.

Figure 6–11 Application Module Data Model in the Business Component Browser



Double-clicking on the view link instance executes the master object and displays the master-detail data in the data view page. For example, in Figure 6–12, double-clicking the ProductWarehousesLevelsLink1 view link instance in the tree executes the ProductsByCategory master view instance in the top portion of the data view page and the ProductStockLevelsByLocation1 view instance in the bottom portion of the data view page. Additional context menu items on the view object node allow you to reexecute the query if needed, remove the view object from the data model panel, and perform other tasks.

In the master-detail data view page, you can scroll through the query results. Additionally, because instances of entity-based view objects are fully editable, instead of displaying *disabled* UI controls showing read-only data for a read-only view object, the data view page displays editable fields. You are free to experiment with creating, inserting, updating, validating, committing, and rolling back.

Figure 6–12 Master-Detail Data View Page in the Business Component Browser

WarehouseId	WarehouseName	ProductId	QuantityOnHand	Address1	City	State/Zip
103	Eastern Seaboard ...	1	1500	100 N Pea...	Philadelphia	PA 19103
102	Main Stuffz.com W...	1	750	615 N She...	Madison	WI 53703
101	Mid-America Ware...	1	200	514 W Su...	Kokomo	IN 46901

For example, you can view multiple levels of master-detail hierarchies, opening multiple data view pages at the same time. Use the **Detach** context menu item to open any tab into a separate window and visualize multiple view object's data at the same time.

Using just the master-detail data view page, you can test several functional areas of your application.

6.3.4.1 Testing Master-Detail Coordination

When you click the navigation buttons on the toolbar, you can see that the rows for the current master view object are correctly coordinated.

6.3.4.2 Testing UI Control Hints

The entity-based view object attributes inherit their control hints from those on the underlying entity object attribute. The prompts displayed in the data view page help you see whether you have correctly defined a user-friendly label text control hint for each attribute. For details on setting up the hint on your entity object, see [Section 5.12, "Defining Attribute Control Hints for View Objects"](#).

6.3.4.3 Testing Business Domain Layer Validation

Depending on the validation rules you have defined, you can try entering invalid values to trigger and verify validation exceptions. For example, when you have defined a range validation rule, enter a value outside the range and see an error similar to:

```
(oracle.jbo.AttrSetValException) Valid product codes are between 100 and 999
```

Click the rollback button in the toolbar to revert data to the previous state.

6.3.4.4 Testing Alternate Language Message Bundles and Control Hints

When your application defines alternative languages in your resource message bundles, you can configure the Business Component Browser to recognize these languages. In the Business Component Browser, you can then display the **Locale** menu and select among the available language choices. To configure the Business Component Browser, select **Preferences** from the JDeveloper **Tools** menu, expand **Business Components** in the selection panel, and select **Tester**. In the Business Components Tester page, add any locale for which you have created a resource message bundle to the **Selected** list.

Alternatively, you can configure the default language choice by setting ADF Business Components runtime configuration properties. These runtime properties also determine which language the Business Component Browser will display as the default. In the Select Business Component Browser Configuration dialog, select the **Properties** tab and enter the desired country code for the country and language. For example, to specify the Italian language, you would enter **IT** for these two properties:

- `jbo.default.country = IT`
- `jbo.default.language = it`

Testing the language message bundles in the Business Component Browser lets you verify that the translations of the entity object control hints are correctly located. Or, if the message bundle defines date formats for specific attributes, the tool lets you verify that date formats change (like 04/12/2007 to 12/04/2007).

6.3.4.5 Testing View Objects That Reference Entity Usages

By scrolling through the data — or using the **Specify View Criteria** button in the Business Component Browser toolbar to search — you can verify whether you have correctly altered the WHERE clause in an entity-based view object's query to use an outer join. The rows should appear as expected.

You also can try changing a primary key attribute of a master view object. This will allow you to verify that the corresponding reference information is automatically updated to reflect the new primary key value.

Use the Business Component Browser to verify that control hints defined at the view object level override the ones it would normally inherit from the underlying entity object. If you notice that several attributes share the same label text, you can edit the control hint for the desired attributes at the view object level. For example, you can set the **Label Text** hint to **Sales Representative Email Address** for the `SalesRepresentativeEmail` attribute and **Customer Email Address** for the `CustomerEmail` attribute.

6.3.4.6 Testing Row Creation and Default Value Generation

When displaying an entity-based view object, click on the **Create Row** button in the Business Component Browser toolbar for the view object instance to create a new blank row. Any fields that have a declarative default value will appear with that value in the blank row. If the a DBSequence-valued attribute is used, a temporary value will appear in the new row. After entering all the required fields, click on the commit button to commit the transaction. The actual, trigger-assigned primary key should appear in the field after successful commit.

6.3.4.7 Testing That New Detail Rows Have Correct Foreign Keys

If you try adding a new row to an existing detail entity-based view object instance, you'll notice that the view link automatically ensures that the foreign key attribute value in the new row is set to the value of the current master service request row.

6.3.5 How to Test Multiuser Scenarios in the Business Component Browser

When view objects and entity objects cooperate at runtime, two exceptions can occur when running the application in a multiuser environment. To anticipate these exceptions, you can simulate a multiuser environment for testing purposes using the Business Component Browser. To accomplish this, open two instances of the Business Component Browser on the application module and do not exit from the first instance. These two tests demonstrate how multiuser exceptions can arise:

- In one instance of the Business Component Browser, modify an attribute of an existing view object and tab out of the field. Then, in the other browser instance, try to modify the same view object attribute in some way. You'll see that the second user gets the `oracle.jbo.AlreadyLockedException`

You can then change the value of `jbo.locking.mode` to be `optimistic` on the Properties page of the Business Component Browser Connect dialog and try repeating the test. You'll see the error occurs at commit time for the second user instead of immediately.

- In one instance of the Business Component Browser, modify an attribute of an existing view object and tab out of the field. Then, in the other browser instance, retrieve (but don't modify) the same view object attribute. Back in the first window, commit the change. If the second user then tries to modify that same attribute, you'll see that the second user gets the

`oracle.jbo.RowInconsistentException`. The row has been modified and committed by another user since the second user retrieved the row into the entity cache.

6.3.6 How to Customize Configuration Options Before Running the Browser

Using the Select Business Components Configuration dialog, you can select a predefined configuration to run the tool using that named set of runtime configuration properties. The Select Configuration dialog also features a **Properties** tab that allows you to see the selected configurations settings and to override any of the configuration's settings for the current run of the browser. For example, you could alter the default language for the UI control hints for a single instance of the Business Component Browser by opening the **Properties** tab and setting the following two properties with the desired country code (in this case, IT for Italy):

- `jbo.default.country = IT`
- `jbo.default.language = it`

Tip: If you wanted to make the changes to your configuration permanent, you could use the Configuration Manager to copy the current configuration and create a new configuration in which you set the desired properties set. For example, anytime you wanted to test in Italian you could simply choose to use the `UserServiceLocalItalian` configuration, instead of the default `UserServiceLocal`.

6.3.7 How to Enable ADF Business Components Debug Diagnostics

When launching the Business Component Browser, if your data model project's current run configuration is set to include the Java System parameter `jbo.debugoutput=console`, you can enable ADF Business Components debug diagnostics with messages directed to the JDeveloper Log window.

Note: Despite the similar name, the JDeveloper project's run configurations are different from the ADF application module's configurations. The former are part of the project properties, the latter are defined along with your application module component in its `bc4j.xcfg` file and edited using the configuration editor.

To set the system debug output property, open the Run/Debug page in the Project Properties dialog for your data model project. Click **Edit** to edit the chosen run configuration, and add following string to the **Java Options** field in the page.

`-Djbo.debugoutput=console`

The next time you run the Business Component Browser and double-click on the view object, you'll see detailed diagnostic output in the console, as shown in [Example 6–1](#). Using the diagnostics will allow you to visualize everything the framework components are doing for your application.

Example 6–1 Diagnostic Output of Business Component Browser

```
:  
[234] Created root application module: 'devguide.examples.UserService'  
[235] Stringmanager using default locale: 'en_US'  
[236] Locale is: 'en_US'
```

```
[237] ApplicationPoolImpl.resourceStateChanged wasn't release related.
[238] Oracle SQLBuilder: Registered driver: oracle.jdbc.driver.OracleDriver
[239] Creating a new pool resource
[240] Trying connection/2: url='jdbc:oracle:thin:@localhost:1521:XE' ...
[241] Successfully logged in
[242] JDBCVersion: 10.1.0.5.0
[243] DatabaseProductName: Oracle
[244] DatabaseProductVersion: Oracle Database 10g Release 10.2.0.1.0
[245] Column count: 4
[246] ViewObject: UserList Created new QUERY statement
[247] UserList>#q computed SQLStmtBufLen: 110, actual=70, storing=100
[248] select USER_ID, EMAIL, FIRST_NAME, LAST_NAME
from USERS
order by EMAIL
:
```

Other legal values for this property are `silent` (the default, if not specified) and `file`. If you choose the file option, diagnostics are written to the system `temp` directory.

Tip: You can create separate JDeveloper run configurations, one with the ADF Business Components debug diagnostics enabled, and another without it. By choosing the appropriate project run configuration, you can easily run JDeveloper with or without debug diagnostics.

6.3.8 What Happens at Runtime: When View Objects and Entity Objects Cooperate

On their own, view objects and entity objects simplify two important jobs that every enterprise application developer needs to do:

- Work with SQL query results
- Modify and validate rows in database tables

Entity-based view objects can query any selection of data that you want the end user to be able to view and modify. Any data the end user is allowed to change will be validated and saved by your reusable business domain layer. The key ingredients you provide as the developer are the ones that only *you* can know:

- You decide what business logic should be enforced in your business domain layer
- You decide what queries describe the data you need to put on the screen

These are the things that make *your* application unique. The built-in functionality of your entity-based view objects handles the rest of the implementation details.

Note: Understanding row keys and what role the entity cache plays in the transaction are important concepts that help to clarify the nature of the entity-based view objects. These two concepts are addressed in [Section 6.4.1, "ViewObject Interface Methods for Working with the View Object's Default RowSet"](#).

6.3.8.1 What Happens When a View Object Executes Its Query

After adding an instance of an entity-based view object to the application module's data model, you can see what happens at runtime when you execute the query. Like a read-only view object, an entity-based view object sends its SQL query straight to the database using the standard Java Database Connectivity (JDBC) API, and the database

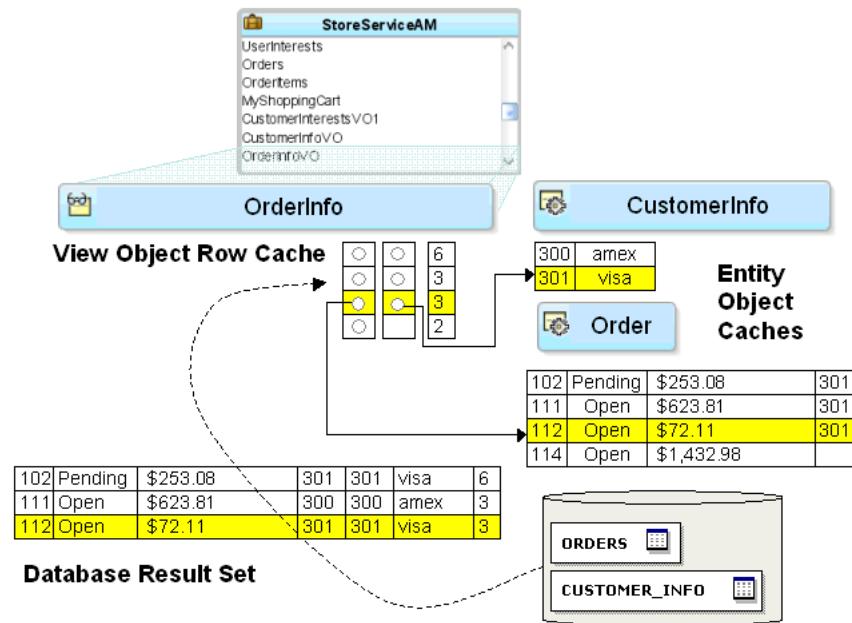
produces a result set. In contrast to its read-only counterpart, however, as the entity-based view object retrieves each row of the database result set, it partitions the row attributes based on which entity usage they relate to. This partitioning occurs by creating an entity object row of the appropriate type for each of the view object's entity usages, populating them with the relevant attributes retrieved by the query, and storing each of these entity rows in its respective entity cache. Then, rather than storing duplicate copies of the data, the view row simply *points* at the entity row parts that comprise it.

As shown in [Figure 6–13](#), the highlighted row in the result set is partitioned into an Order entity row with primary key 112 and a CustomerInfo entity row with primary key 301.

As described in [Section 6.4.1.2, "The Role of the Entity Cache in the Transaction"](#), the entity row that is brought into the cache using `findByPrimaryKey()` contains *all* attributes of the entity object. In contrast, an entity row created by partitioning rows from the entity-based view object's query result contains values only for attributes that appear in the query. It does *not* include the complete set of attributes. This partially populated entity row represents an important runtime performance optimization.

Since the ratio of rows retrieved to rows modified in a typical enterprise application is very high, you can save memory by bringing only the attributes into memory that you need to display instead of bringing all attributes into memory all the time.

Figure 6–13 Entity Cache Partitions View Rows into Entity Rows



By partitioning queried data this way into its underlying entity row constituent parts, the first benefit you gain is that all of the rows that include some data queried will display a consistent result when changes are made in the current transaction. In other words, if one view object allows the `PaymentType` attribute of customer 301 to be modified, then all rows in any entity-based view object showing the `PaymentType` attribute for customer 301 will update instantly to reflect the change. Since the data related to customer 301 is stored exactly once in the `CustomerInfo` entity cache in the entity row with primary key 301, any view row that has queried the order's `PaymentType` attribute is just pointing at this single entity row.

Luckily, these implementation details are completely hidden from a client working with the rows in a view object's row set. The client works with a view row, getting and setting the attributes, and is unaware of how those attributes might be related to entity rows behind the scenes.

6.3.8.2 What Happens When a View Row Attribute Is Modified

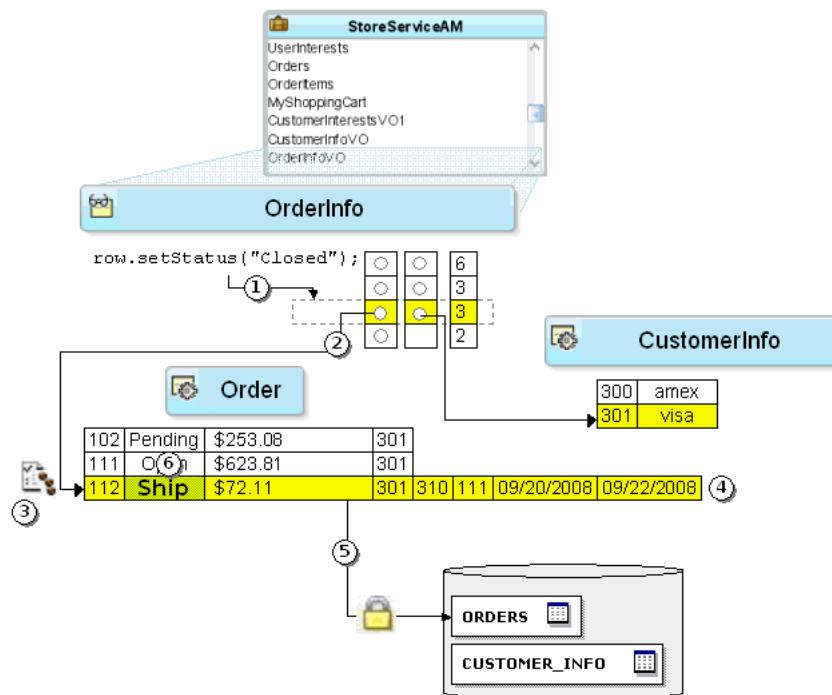
When a user attempts to update the attribute of a view row, a series of steps occur to automatically coordinate this view row attribute modification with the underlying entity row. [Figure 6-14](#) illustrates one example of an update attempt made by the user and the steps that will occur:

1. The order processor attempts to set the `Status` attribute to the value `Ship`.
2. Since `Status` is an entity-mapped attribute from the `Order` entity usage, the view row delegates the attribute set to the appropriate underlying entity row in the `Order` entity cache having primary key 112.
3. Any attribute-level validation rules on the `Status` attribute of the `Order` entity object are evaluated and the modification attempt will fail if any rule does not succeed.

Assume that some validation rule for the `Status` attribute programmatically references the `ShipDate` attribute (for example, to enforce a business rule that an `Order` cannot be shipped the same day it is placed). The `ShipDate` was not one of the `Order` attributes retrieved by the query, so it is not present in the partially populated entity row in the `Order` entity cache.

4. To ensure that business rules can always reference all attributes of the entity object, the entity object detects this situation and "faults-in" the entire set of `Order` entity object attributes for the entity row being modified using the primary key (which must be present for each entity usage that participates in the view object).
5. After the attribute-level validations all succeed, the entity object attempts to acquire a lock on the row in the `ORDERS` table before allowing the first attribute to be modified.
6. If the row can be locked, the attempt to set the `Status` attribute in the row succeeds and the value is changed in the entity row.

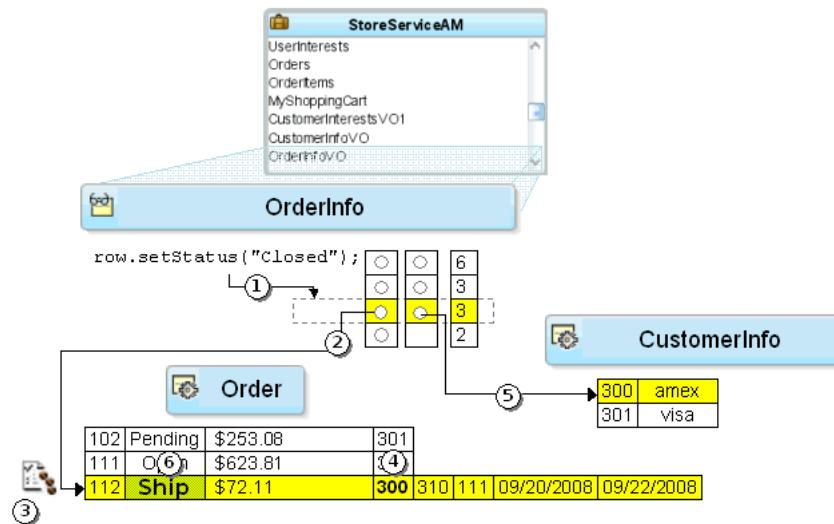
Note: The `jbo.locking.mode` configuration property controls how rows are locked. The default value is `pessimistic`. In `pessimistic` locking mode, the row must be lockable before any change is allowed to it in the entity cache. For simplicity, this is the behavior that this example describes. However, typically, Fusion web applications will set this property to `optimistic` instead, so that rows aren't locked until transaction commit time.

Figure 6–14 View Row Attribute Updates Delegate to the Entity

6.3.8.3 What Happens When a Foreign Key Attribute is Changed

When a user attempts to update a foreign key attribute, a series of steps occur to automatically coordinate this view row attribute modification with the underlying entity row. [Figure 6–15](#) illustrates one example of an update attempt made by the user and the steps that will occur:

1. The order processor attempts to set the `CustomerInfoId` attribute to the value 300.
2. Since `CustomerInfoId` is an entity-mapped attribute from the `Order` entity usage, the view row delegates the attribute set to the appropriate underlying entity row in the `Order` entity cache, which has primary key 112.
3. Any attribute-level validation rules on the `CustomerInfoId` attribute of the `Order` entity object are evaluated and the modification attempt will fail if any rule does not succeed.
4. The row is already locked, so the attempt to set the `CustomerInfoId` attribute in the row succeeds and the value is changed in the entity row.
5. Since the `CustomerInfoId` attribute on the `Order` entity usage is associated with the `CustomerInfo` entity object, this change of foreign key value causes the view row to replace its current entity row part for customer 301 with the entity row corresponding to the new `CustomerInfoId = 300`. This effectively makes the view row for order 112 point to the entity row for 300, so the value of the `PaymentType` in the view row updates to reflect the correct reference information for this newly assigned customer.

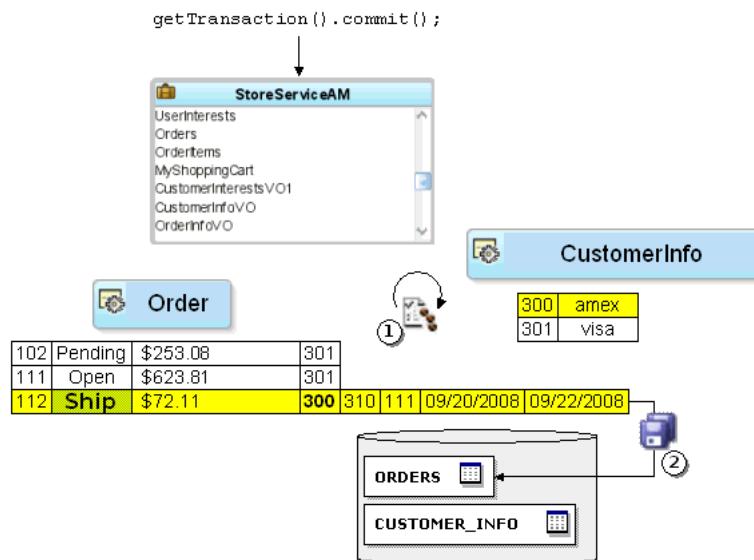
Figure 6–15 After Updating a Foreign Key, View Row Points to a New Entity

6.3.8.4 What Happens When a Transaction is Committed

Suppose the user is satisfied with the changes, and commits the transaction. As shown in **Figure 6–16**, there are two basic steps:

1. The Transaction object validates any invalid entity rows in its pending changes list.
2. The entity rows in the pending changes list are saved to the database.

The figure depicts a loop in Step 1 before the act of validating one modified entity object might programmatically affect changes to other entity objects. Once the transaction has processed its list of invalid entities on the pending changes list, if the list has entities, the transaction will complete another pass. It will attempt up to ten passes through the list. If by that point there are still invalid entity rows, it will throw an exception because this typically means you have an error in your business logic that needs to be investigated.

Figure 6–16 Committing the Transaction Validates Invalid Entities, Then Saves Them

6.3.8.5 What Happens When a View Object Requeries Data

When you reexecute a view object's query, by default the view rows in its current row set are "forgotten" in preparation for reading in a fresh result set. This view object operation does not directly affect the entity cache, however. The view object then sends the SQL to the database and the process begins again to retrieve the database result set rows and partition them into entity row parts.

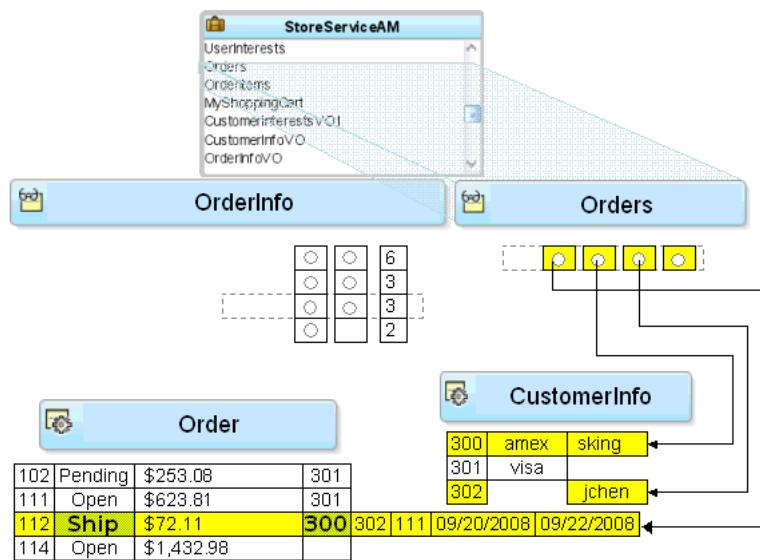
Note: Typically when the view object requeries data, you expect it to retrieve the latest database information. If instead you want to avoid a database roundtrip by restricting your view object to querying only over existing entity rows in the cache, or over existing rows already in the view object's row set, see [Section 35.6, "Performing In-Memory Sorting and Filtering of Row Sets"](#).

6.3.8.5.1 How Unmodified Attributes are Handled During Requery As part of the entity row partitioning process during a requery, if an attribute on the entity row is unmodified, then its value in the entity cache is updated to reflect the newly queried value.

6.3.8.5.2 How Modified Attributes are Handled During Requery However, if the value of an entity row attribute has been *modified* in the current transaction, then during a requery the entity row partitioning process does not refresh its value. Uncommitted changes in the current transaction are left intact so the end-user's logical unit of work is preserved. As with any entity attribute value, these pending modifications continue to be consistently displayed in any entity-based view object rows that reference the modified entity rows.

Note: End-user row inserts and deletes are also managed by the entity cache, which permits new rows to appear and deleted rows to be skipped during requerying. For more information about new row behavior, see [Section 35.1.2, "Consistently Displaying New Rows in View Objects Based on the Same Entity"](#).

For example, [Figure 6–17](#) illustrates the scenario where a user "drills down" to a different page that uses the `Orders` view object instance to retrieve all details about order 112 and that this happens in the context of the current transaction's pending changes. That view object has two entity usages: a primary `Orders` usage and a reference usage for `CustomerInfo`. When its query result is partitioned into entity rows, it ends up pointing at the same `Order` entity row that the previous `OrderInfo` view row had modified. This means the end user will correctly see the pending change, that the order is assigned to skinning in this transaction.

Figure 6–17 Entity Cache Merges Sets of Entity Attributes from Different View Objects

6.3.8.5.3 How Overlapping Subsets of Attributes are Handled During Requery Two different view objects can retrieve two *different* subsets of reference information and the results are merged whether or not they have matching sets of attributes. For example, Figure 6–17 also illustrates the situation, where the ServiceRequests queries up FirstName and LastName for a user, while the OpenProblemsAndAssignees view object queried the user's Email. The figure shows what happens at runtime: if while partitioning the retrieved row, the entity row part contains a different set of attributes than does the partially populated entity row that is already in the cache, the attributes get "merged". The result is a partially populated entity row in the cache with the *union* of the overlapping subsets of user attributes. In contrast, for John Chen (user 308), who wasn't in the cache already, the resulting new entity row contains only the FirstName and LastName attributes, but not the Email.

6.3.9 What You May Need to Know About Optimizing View Object Runtime Performance

The view object provides tuning parameters that let you control how SQL is executed and how data is fetched from the database. These tuning parameters play a large role in the runtime performance of the view object. If the fetch options are not tuned correctly for the application, then your view object may fetch an excessive amount of data and may make too many roundtrips to the database.

You can use the **Tuning** section of the General page of the overview editor to configure the fetch options shown in Table 6–1.

Table 6–1 Parameters to Tune View Object Performance

LOV List Type	Usage
Fetch Mode	The default fetch option is the All Rows option, which will be retrieved As Needed (<code>FetchMode="FETCH_AS_NEEDED"</code>) or All at Once (<code>FetchMode="FETCH_ALL"</code>), depending on which option is desired. The As Needed option ensures that an <code>executeQuery()</code> operation on the view object initially retrieves only as many rows as necessary to fill the first page of a display, whose number of rows is set based on the view object's range size.
Fetch Size	In conjunction with the Fetch Mode option, the in Batches of field controls the number of records fetched at one time from the database (<code>FetchSize</code> in the view object XML). The default value is 1, which will give poor performance unless only one row will be fetched. The suggested configuration is to set this value to $n+1$ where n is the number of rows to be displayed in the user interface.
Max Fetch Size	The default max fetch size for a view object is -1, which means that there is no limit to the number of rows the view object can fetch. In cases where the result set should contain only n rows of data, the option Only up to row number should be selected and set to n . The developer can alternatively call <code>setMaxFetchSize(n)</code> to set this programmatically or manually add the parameter <code>MaxFetchSize</code> to the view object XML.
	For view objects whose WHERE clause expects to retrieve a single row, set the option At Most One Row . This way the view object knows you don't expect any more rows and it will skip its normal test for that situation.
	As mentioned earlier, setting a maximum fetch size of 0 (zero) makes the view object insert-only. In this case, no select query will be issued, so no rows will be fetched.
Forward-only Mode	If a data set will only be traversed going forward, then forward-only mode can help performance when iterating through the data set. This can be configured by programmatically calling <code>setForwardOnly(true)</code> on the view object. Setting forward-only will also prevent caching previous sets of rows as the data set is traversed.

When you tune view objects, you should also consider these issues:

- Large data sets: View objects provide a mechanism to page through large data sets such that a user can jump to a specific page in the results. This is configured by calling `setRangeSize(N)` followed by `setAccessMode(RowSet.RANGE_PAGING)` on the view object where N is the number of rows contained within one page. When the user navigates to a specific page in the data set, the application can call `scrollToRangePage(P)` on the view object to navigate to page P .

Range paging fetches and caches only the current page of rows in the view object row cache at the cost of another query execution to retrieve each page of data. Range paging is not appropriate where it is beneficial to have all fetched rows in the view object row cache (for example, when the application needs to read all rows in a dataset for an LOV or page back and forth in records of a small data set).

- Spillover: There is a facility to use the data source as "virtual memory" when the JVM container runs out of memory. By default, this is disabled and can be turned on as a last resort by setting `jbo.use.pers.coll=true`. Enabling spillover can have a large performance impact.
- SQL style: If the generic SQL92 SQL style is used to connect to generic SQL92-compliant databases, then some view object tuning options will not function correctly. The parameter that choosing the generic SQL92 SQL style affects the most is the fetch size. When SQL92 SQL style is used, the fetch size defaults to 10 rows regardless of what is configured for the view object. You can set the SQL style when you define the database connection. By default, the SQL style will be Oracle. To manually override the SQL style, you can also pass the parameter `-Djbo.SQLBuilder="SQL92"` to the JVM upon startup.

Additionally, you have some options to tune the view objects' associated SQL for better database performance:

- Bind variables: If the query associated with the view object contains values that may change from execution to execution, use bind variables. Using bind variables in the query allows the query to reexecute without needing to reparse the query on the database. You can add bind variables to the view object in the Query page of the overview editor for the view object. For more information, see [Section 5.9, "Working with Bind Variables"](#).
- Query optimizer hints: The view object can pass hints to the database to influence which execution plan to use for the associated query. The optimizer hints can be specified in the **Retrieve from the Database** group box in the Tuning section of the overview editor for the view object. For information about optimizer hints, see [Section 35.2.4.3, "Specify a Query Optimizer Hint if Necessary"](#).

6.4 Testing View Object Instances Programmatically

When you are ready to test a working application module containing at least one view object instance, you can build a simple test client program to illustrate the basics of working programmatically with the data in the contained view object instances.

From the point of view of a client accessing your application module's data model, the API's to work with a read-only view object and an entity-based view object are identical. The key functional difference is that entity-based view objects allow the data in a view object to be fully updatable. The application module that contains the entity-based view objects defines the unit of work and manages the transaction. This section presents four simple test client programs that work with the SRService application module to illustrate:

- Iterating master-detail-detail hierarchy
- Finding a row and updating a foreign key value
- Creating a new service request
- Retrieving the row key identifying a row

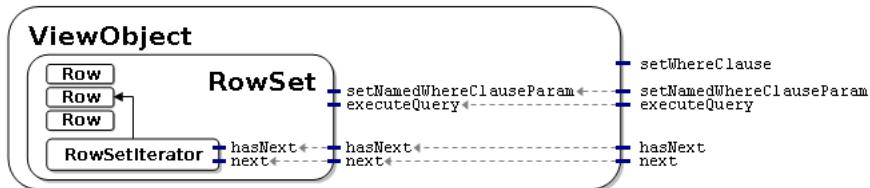
6.4.1 ViewObject Interface Methods for Working with the View Object's Default RowSet

The `ViewObject` interface in the `oracle.jbo` package provides the methods to easily perform any data-retrieval task. Some of these methods used in the example include:

- `executeQuery()`, to execute the view object's query and populate its row set of results
- `setWhereClause()`, to add a dynamic predicate at runtime to narrow a search
- `setNamedWhereClauseParam()`, to set the value of a named bind variable
- `hasNext()`, to test whether the row set iterator has reached the last row of results
- `next()`, to advance the row set iterator to the next row in the row set
- `getEstimatedRowCount()`, to count the number of rows a view object's query would return

Typically, when you work with a view object, you will work with only a single row set of results at a time. To simplify this overwhelmingly common use case, as shown in [Figure 6–18](#), the view object contains a default `RowSet`, which, in turn, contains a default `RowSetIterator`. The default `RowSetIterator` allows you to call all of the data-retrieval methods directly on the `ViewObject` component itself, knowing that they will apply automatically to its default row set.

Figure 6–18 ViewObject Contains a Default RowSet and RowSetIterator



Note: [Chapter 35, "Advanced View Object Techniques"](#) presents situations when you might want a single view object to produce *multiple* distinct row sets of results. You can also find scenarios for creating *multiple* distinct row set iterators for a row set. Most of the time, however, you'll need only a single iterator.

The phrase "working with the rows in a view object," when used in this guide more precisely means working with the rows in the view object's default row set. Similarly, the phrase "iterate over the rows in a view object," more precisely means you will use the default row set iterator of the view object's default row set to loop over its rows.

6.4.1.1 The Role of the Key Object in a View Row or Entity Row

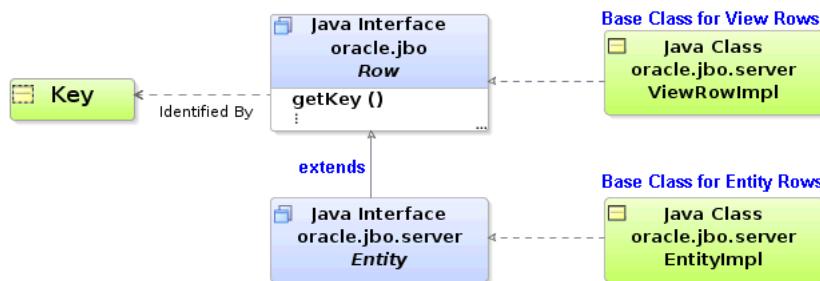
When you work with view rows you use the `Row` interface in the `oracle.jbo` package. As shown in [Figure 6–19](#), the interface contains a method called `getKey()` that you can use to access the `Key` object that identifies any row. Notice that the `Entity` interface in the `oracle.jbo.server` package extends the `Row` interface. This relationship provides a concrete explanation of why the term *entity row* is so appropriate. Even though an entity row supports additional features for encapsulating business logic and handling database access, you can still treat any entity row as a `Row`.

An entity-based view object delegates the task of finding rows by key to its underlying entity row parts.

Recall that both view rows and entity rows support either single-attribute or multiattribute keys, so the Key object related to any given Row will encapsulate all of the attributes that comprise its key. Once you have a Key object, you can use the `findByPrimaryKey()` method on any row set to find a row based on its Key object. When you use the `findByPrimaryKey()` method to find a view row by key, the view row proceeds to use the entity definition's `findByPrimaryKey()` method to find each entity row contributing attributes to the view row key.

In the case of a read-only view object with no underlying entity row to which to delegate this task, the view object implementation automatically enables the `manageRowsByKey` flag when at least one primary key attribute is detected. This ensures that the `findByPrimaryKey()` method is successful in the case of read-only view objects. If the `manageRowsByKey` flag is not enabled, then UI operations like setting the current row with the key, which depend on the `findByPrimaryKey()` method, would not work.

Figure 6–19 Any View Row or Entity Row Supports Retrieving Its Identifying Key



Note: When you define an entity-based view object, by default the primary key attributes for all of its entity usages are marked with their **Key Attribute** property set to `true`. In any *nonupdatable* reference entity usages, you should disable the **Key Attribute** property for the key attributes. Since view object attributes related to the primary keys of *updatable* entity usages must be part of the composite view row key, their **Key Attribute** property cannot be disabled.

6.4.1.2 The Role of the Entity Cache in the Transaction

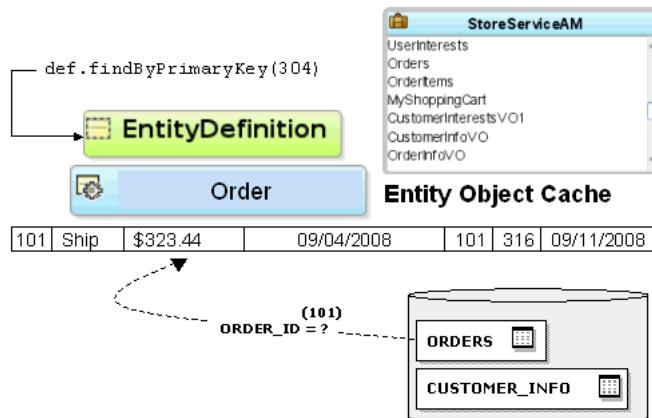
An application module is a transactional container for a logical unit of work. At runtime, it acquires a database connection using information from the named configuration you supply, and it delegates transaction management to a companion `Transaction` object. Since a logical unit of work may involve finding and modifying multiple entity rows of different types, the `Transaction` object provides an entity cache as a "work area" to hold entity rows involved in the current user's transaction. Each entity cache contains rows of a single entity type, so a transaction involving two or more entity objects holds the working copies of those entity rows in separate caches.

By using an entity object's related entity definition, you can write code in an application module to find and modify existing entity rows. As shown in [Figure 6–20](#), by calling `findByPrimaryKey()` on the entity definition for the `Order` entity object, you can retrieve the row with that key. If it is not already in the entity cache, the entity object executes a query to retrieve it from the database. This query selects all of the entity object's persistent attributes from its underlying table, and finds the row using

an appropriate WHERE clause against the column corresponding to the entity object's primary key attribute. Subsequent attempts to find the same entity row by key during the same transaction will find it in the cache, preventing the need for a trip to the database. In a given entity cache, entity rows are indexed by their primary key. This makes finding an entity row in the cache a fast operation.

When you access related entity rows using association accessor methods, they are also retrieved from the entity cache. If related entity rows are not in the cache, then they are retrieved from the database. Finally, the entity cache is also the place where new entity rows wait to be saved. In other words, when you use the `createInstance2()` method on the entity definition to create a new entity row, it is added to the entity cache.

Figure 6–20 Entity Cache Stores Entity Rows During the Transaction



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

The `Transaction` object used by an application module represents the working set of entity rows for a single end-user transaction. By design, it is *not* a shared, global cache. The database engine itself is an extremely efficient shared, global cache for multiple, simultaneous users. Rather than attempting to duplicate all the work of fine-tuning that has gone into the database's shared, global cache functionality, ADF Business Components consciously embraces it. To refresh a single entity object's data from the database at any time, you can call its `refresh()` method. You can `setClearCacheOnCommit()` or `setClearCacheOnRollback()` on the `Transaction` object to control whether entity caches are cleared at commit or rollback. The defaults are `false` and `true`, respectively. The `Transaction` object also provides a `clearEntityCache()` method you can use to programmatically clear entity rows of a given entity type (or all types). When you clear an entity cache, you allow entity rows of that type to be retrieved from the database fresh the next time they are either found by primary key or retrieved by an entity-based view object.

6.4.2 How to Create a Command-Line Java Test Client

To create a test client program, use the Create Java Class wizard, which is accessible from the New Gallery.

To create a command-line Java test client:

1. In the Application Navigator, right-click the project in which you want to create the test client and choose **New**.
2. In the New Gallery, expand **General**, select the **Simple Files** category, and then **Java Class**, and click **OK**.
3. In the Create Java Class dialog, enter a class name, like `TestClient`, a package name, like `oracle.fodemo.storefront.client`, and ensure that the **Extends** field shows `java.lang.Object`.
4. In **Optional Attributes**, deselect **Generate Default Constructor** and select **Generate Main Method**.
5. Click **OK**.

The `.java` file opens in the source editor to show the skeleton code, as shown in [Example 6–2](#).

Example 6–2 Skeleton Code for `TestClient.java`

```
package oracle.fodemo.storefront.client;
public class TestClient {
    public static void main(String[] args) {
        }
}
```

You can proceed to edit the file using the predefined `bc4jclient` code template available from JDeveloper.

To insert the `bc4jclient` code template:

1. Place the cursor on a blank line inside the body of the `main()` method and use the `bc4jclient` code template to create the few lines of necessary code.
2. Type the characters `bc4jclient` followed **Ctrl + Enter**.
JDeveloper will expand the class file with the template as shown in [Example 6–3](#).
3. Adjust the values of the `amDef` and `config` variables to reflect the names of the application module definition and the configuration that you want to use, respectively.

For the [Example 6–3](#), the changed lines look like this:

```
String amDef = "oracle.fodemo.storefront.store.service.StoreServiceAM";
String config = "StoreServiceAMLocal";
```

4. Finally, change the view object instance name in the call to `findViewObject()` to the one you want to work with. Specify the name exactly as it appears in the **Data Model** tree on the Data Model page of the overview editor for the application module.

For the [Example 6–3](#), the changed line looks like this:

```
ViewObject vo = am.findViewObject("Persons");
```

Example 6–3 Expanded Skeleton Code for TestClient.java

```
package oracle.fodemo.storefront.client;
import oracle.jbo.client.Configuration;
import oracle.jbo.*;
import oracle.jbo.domain.Number;
import oracle.jbo.domain.*;
public class TestClient {
    public static void main(String[] args) {
        String      amDef = "test.TestModule";
        String      config = "TestModuleLocal";
        ApplicationModule am =
            Configuration.createRootApplicationModule(amDef,config);
        ViewObject vo = am.findViewObject("TestView");
        // Work with your appmodule and view object here
        Configuration.releaseRootApplicationModule(am,true);
    }
}
```

Your skeleton test client for your application module should contain source code like what you see in [Example 6–4](#).

Note: The examples throughout [Section 9.10, "Working Programmatically with an Application Module's Client Interface"](#) expand this test client sample code to illustrate calling custom application module service methods, too.

Example 6–4 Working Skeleton Code for an Application Module Test Client Program

```
package oracle.fodemo.storefront.client;
import oracle.jbo.ApplicationModule;
import oracle.jbo.Row;
import oracle.jbo.RowSet;
import oracle.jbo.ViewObject;
import oracle.jbo.client.Configuration;

public class TestClient {

    public static void main(String[] args) {
        String  amDef = "oracle.fodemo.storefront.store.service.StoreServiceAM";
        String  config = "StoreServiceAMLocal";
        ApplicationModule am =
            Configuration.createRootApplicationModule(amDef,config);
        // 1. Find the Persons view object instance.
        ViewObject personList = am.findViewObject("Persons");
        // Work with your appmodule and view object here
        Configuration.releaseRootApplicationModule(am,true);
    }
}
```

To execute the view object's query, display the number of rows it will return, and loop over the result to fetch the data and print it out to the console, replace `// Work with your appmodule and view object here`, with the code in [Example 6–5](#)

Example 6–5 Looping Over Master-Detail View Objects and Printing the Results to the Console

```
// 2. Execute the query
personList.executeQuery();
```

```

// 3. Iterate over the resulting rows
while (personList.hasNext()) {
    Row person = personList.next();
    // 4. Print the person's email
    System.out.println("Person: " + person.getAttribute("Email"));
    // 5. Get related rowset of Orders using view link accessor attribute
    RowSet orders = (RowSet)person.getAttribute("Orders");
    // 6. Iterate over the Orders rows
    while (orders.hasNext()) {
        Row order = orders.next();
        // 7. Print out some order attribute values
        System.out.println(" ["+order.getAttribute("OrderStatusCode")+"] "+ order.getAttribute("OrderId")+": "+ order.getAttribute("OrderTotal"));
        if(!order.getAttribute("OrderStatusCode").equals("COMPLETE")) {
            // 8. Get related rowset of OrderItems
            RowSet items = (RowSet)order.getAttribute("OrderItems");
            // 9. Iterate over the OrderItems rows
            while (items.hasNext()) {
                Row item = items.next();
                // 10. Print out some order items attributes
                System.out.println(" "+item.getAttribute("LineItemId")+": "+ item.getAttribute("LineItemTotal"));
            }
        }
    }
}

```

The first line calls the `executeQuery()` method to execute the view object's query. This produces a row set of zero or more rows that you can loop over using a `while` statement that iterates until the view object's `hasNext()` method returns `false`. Inside the loop, the code puts the current `Row` in a variable named `person`, then invokes the `getAttribute()` method twice on that current `Row` object to get the value of the `Email` and `Orders` attributes to print order information to the console. A second `while` statement performs the same task for the line items of the order.

6.4.3 What Happens When You Run a Test Client Program

The call to `createRootApplicationModule()` on the `Configuration` object returns an instance of the application module to work with. As you might have noticed in the debug diagnostic output, the ADF Business Components runtime classes load XML component definitions as necessary to instantiate the application module and the instance of the view object component that you've defined in its data model at design time. The `findViewObject()` method on the application module finds a view object instance by name from the application module's data model. After the loop shown in [Example 6–5](#), the test client executes `releaseRootApplicationModule()` on the `Configuration` object. This signals that you're done using the application module and it allows the framework to clean up resources, like the database connection that was used by the application module.

6.4.4 What You May Need to Know About Running a Test Client

The `createRootApplicationModule()` and `releaseRootApplicationModule()` methods are very useful for command-line access to application module components. However, you typically won't need to write these two lines of code in the context of an ADF-based web or Swing application. The ADF Model data binding layer cooperates automatically with the ADF Business

Components layer to acquire and release application module components for you in those scenarios.

6.4.5 How to Count the Number of Rows in a Row Set

The `getEstimatedRowCount()` method is used on a `RowSet` to determine how many rows it contains:

```
long numReqs = reqs.getEstimatedRowCount();
```

The implementation of the `getEstimatedRowCount()` initially issues a `SELECT COUNT(*)` query to calculate the number of rows that the query will return. The query is formulated by "wrapping" your view object's entire query in a statement like:

```
SELECT COUNT(*) FROM ( ... your view object's SQL query here ... )
```

The `SELECT COUNT(*)` query allows you to access the count of rows for a view object without necessarily retrieving all the rows themselves. This approach permits an important optimization for working with queries that return a large number of rows, or for testing how many rows a query *would* return before proceeding to work with the results of the query.

Once the estimated row count is calculated, subsequent calls to the method do not reexecute the `COUNT(*)` query. The value is cached until the next time the view object's query is executed, since the fresh query result set returned from the database could potentially contain more, fewer, or different rows compared with the last time the query was run. The estimated row count is automatically adjusted to account for pending changes in the current transaction, adding the number of relevant new rows and subtracting the number of removed rows from the count returned.

You can also override `getEstimatedRowCount()` to perform a custom count query that suits your application's needs.

6.4.6 How to Access a Detail Collection Using the View Link Accessor

Once you've retrieved the `RowSet` of detail rows using a view link accessor, as described in [Section 5.6.6.2, "Programmatically Accessing a Detail Collection Using the View Link Accessor"](#), you can loop over the rows it contains using the same pattern used by the view object's row set of results:

```
while (reqs.hasNext()) {  
    Row curReq = reqs.next();  
    System.out.println("--> (" + curReq.getAttribute("SvrId") + ") " +  
        curReq.getAttribute("ProblemDescription"));  
}
```

[Example 6–6](#) shows the `main()` method sets a dynamic WHERE clause to restrict the `PersonList` view object instance to show only persons whose `person_type_code` has the value `CUST`. Additionally, the `executeAndShowResults()` method accesses the view link accessor attribute and prints out the request number (`PersonId`) and `Email` attribute for each one.

If you use JDeveloper's **Refactor > Duplicate** functionality on the existing `TestClient.java` class (see [Example 6–7](#)), you can quickly "clone" it to create a `TestClient2.java` class that you'll modify as shown in [Example 6–6](#) to make this technique.

Performance Tip: If the code you write to loop over the rows does not need to display them, then you can call the `closeRowSet()` method on the row set when you're done. This technique will make memory use more efficient. The next time you access the row set, its query will be reexecuted.

Example 6–6 Programmatically Accessing Detail Rows Using the View Link Accessor

```
package devguide.examples.readonlyvo.client;
import oracle.jbo.ApplicationModule;
import oracle.jbo.Row;
import oracle.jbo.RowSet;
import oracle.jbo.ViewObject;
import oracle.jbo.client.Configuration;

public class TestClient2 {
    public static void main(String[] args) {
        String amDef = "devguide.examples.readonlyvo.PersonService";
        String config = "PersonServiceLocal";
        ApplicationModule am =
            Configuration.createRootApplicationModule(amDef, config);
        ViewObject vo = am.findViewObject("PersonList");
        // Add an extra where clause with a new named bind variable
        vo.setWhereClause("person_type_code = :ThePersonType");
        vo.defineNamedWhereClauseParam("ThePersonType", null, null);
        vo.setNamedWhereClauseParam("ThePersonType", "CUST");
        // Show results when :ThePersonType = 'CUST'
        executeAndShowResults(vo);
        Configuration.releaseRootApplicationModule(am, true);
    }
    private static void executeAndShowResults(ViewObject vo) {
        System.out.println("---");
        vo.executeQuery();
        while (vo.hasNext()) {
            Row curPerson = vo.next();
            // Access the row set of details using the view link accessor attribute
            RowSet orders = (RowSet)curPerson.getAttribute("OrdersShippedToPurchaser");
            long numOrders = orders.getEstimatedRowCount();
            System.out.println(curPerson.getAttribute("PersonId") + " " +
                curPerson.getAttribute("Email")+" ["+
                numOrders+" orders]");
            while (orders.hasNext()) {
                Row curOrder = orders.next();
                System.out.println("--> (" + curOrder.getAttribute("OrderId") + ") " +
                    curOrder.getAttribute("OrderTotal"));
            }
        }
    }
}
```

Running `TestClient2` shows the following results in the Log window. Each customer is listed, and for each customer that has some orders, the order total appears beneath their name.

```
---
121 AFRIPP [12 orders]
115 AKHOO [12 orders]
109 DFAVIET [12 orders]
114 DRAPHEAL [12 orders]
118 GHIMURO [12 orders]
```

```
126 IMIKKILI [12 orders]
111 ISCIARRA [12 orders]
110 JCHEN [12 orders]
127 JLANDRY [12 orders]
112 JMURMAN [12 orders]
125 JNAYER [12 orders]
119 KCOLMENA [12 orders]
124 KMOURGOS [12 orders]
129 LBISSOT [12 orders]
113 LPOPP [12 orders]
--> (1013) 89.99
120 MWEISS [12 orders]
--> (1003) 5000
108 NGREENBE [12 orders]
--> (1002) 1249.91
122 PKAUFLIN [12 orders]
116 SBAIDA [12 orders]
128 SMARKLE [12 orders]
117 STOBIAS [12 orders]
123 SVOLLMAN [12 orders]
```

If you run `TestClient2` with debug diagnostics enabled, you will see the SQL queries that the view object performed. The view link `WHERE` clause predicate is used to automatically perform the filtering of the detail service request rows for the current row in the `UserList` view object.

6.4.7 How to Iterate Over a Master-Detail-Detail Hierarchy

[Example 6–7](#) performs the following basic steps:

1. Finds the `Persons` view object instance.
2. Executes the query.
3. Iterates over the resulting `personList` rows.
4. Prints the person's email address by getting the value of the `Email` attribute.
5. Gets related row set of `Orders` using the view link accessor attribute.
6. Iterates over the `Orders` rows.
7. Prints out some of the `Orders` attribute values.
8. If the status is not `COMPLETE`, then gets the related row set of `OrderItems` using the view link accessor attribute.
9. Iterates over the `OrderItems` rows.
10. Prints out some of the `OrderItems` attributes.

Note: Other than having one additional level of nesting, this example uses the same API's that you saw in the `TestClient2` program that was iterating over master-detail read-only view objects in [Section 6.4.6, "How to Access a Detail Collection Using the View Link Accessor"](#).

Example 6–7 Iterating Master/Detail/Detail Hierarchy

```
package oracle.fodemo.storefront.client;
import oracle.jbo.ApplicationModule;
import oracle.jbo.Row;
```

```

import oracle.jbo.RowSet;
import oracle.jbo.ViewObject;
import oracle.jbo.client.Configuration;

public class TestClient {

    public static void main(String[] args) {
        String amDef =
            "oracle.fodemo.storefront.store.service.StoreServiceAM";
        String config = "StoreServiceAMLocal";
        ApplicationModule am =
            Configuration.createRootApplicationModule(amDef, config);
        // 1. Find the Persons view object instance.
        ViewObject personList = am.findViewObject("Persons");
        // 2. Execute the query
        personList.executeQuery();
        // 3. Iterate over the resulting rows
        while (personList.hasNext()) {
            Row person = personList.next();
            // 4. Print the person's email
            System.out.println("Person: " + person.getAttribute("Email"));
            // 5. Get related rowset of Orders using view link accessor attribute
            RowSet orders = (RowSet)person.getAttribute("Orders");
            // 6. Iterate over the Orders rows
            while (orders.hasNext()) {
                Row order = orders.next();
                // 7. Print out some order attribute values
                System.out.println(" [" + order.getAttribute("OrderStatusCode") + "] " +
                    order.getAttribute("OrderId") + ": " +
                    order.getAttribute("OrderTotal"));
                if (!order.getAttribute("OrderStatusCode").equals("COMPLETE")) {
                    // 8. Get related rowset of OrderItems
                    RowSet items = (RowSet)order.getAttribute("OrderItems");
                    // 9. Iterate over the OrderItems rows
                    while (items.hasNext()) {
                        Row item = items.next();
                        // 10. Print out some order items attributes
                        System.out.println(" " + item.getAttribute("LineItemId") + ": " +
                            item.getAttribute("LineItemTotal"));
                    }
                }
            }
        }
        Configuration.releaseRootApplicationModule(am, true);
    }
}

```

6.4.8 How to Find a Row and Update a Foreign Key Value

Example 6–8 performs the following basic steps:

1. Finds the Order view object instance.
2. Constructs a Key object to look up the row for order number 1011.
3. Uses `findByPrimaryKey()` to find the row.
4. Prints value of an `OrderStatusCode` attribute.
5. Tries to assign the *illegal* value REOPENED to the `OrderStatusCode` attribute

Since view object rows cooperate with entity objects, the validation rule on the OrderStatusCode attribute throws an exception, preventing this illegal change.

6. Sets the OrderStatusCode to a legal value of PENDING.
7. Prints the value of the OrderStatusCode attribute to show it was updated successfully.
8. Prints the current value of the customer for this order.
9. Reassigns the order to customer number 113 by setting the CustomerId attribute.
10. Shows the value of the reference information (CustomerId) reflecting a new customer.
11. Cancels the transaction by issuing a rollback.

Example 6–8 Finding and Updating a Foreign Key Value

```
package oracle.fodemo.storefront.client;

import oracle.jbo.ApplicationModule;
import oracle.jbo.JboException;
import oracle.jbo.Key;
import oracle.jbo.Row;
import oracle.jbo.ViewObject;
import oracle.jbo.client.Configuration;

public class TestFindAndUpdate {
    public static void main(String[] args) {
        String amDef =
            "oracle.fodemo.storefront.store.service.StoreServiceAM";
        String config = "StoreServiceAMLocal";
        ApplicationModule am =
            Configuration.createRootApplicationModule(amDef, config);
        // 1. Find the Orders view object instance
        ViewObject vo = am.findViewObject("Orders");
        // 2. Construct a new Key to find Order # 1011
        Key orderKey = new Key(new Object[]{1011});
        // 3. Find the row matching this key
        Row[] ordersFound = vo.findByKey(orderKey,1);
        if (ordersFound != null && ordersFound.length > 0) {
            Row order = ordersFound[0];
            // 4. Print some order information
            String orderStatus = (String)order.getAttribute("OrderStatusCode");
            System.out.println("Current status is: " + orderStatus);
            try {
                // 5. Try setting the status to an illegal value
                order.setAttribute("OrderStatusCode", "REOPENED");
            }
            catch (JboException ex) {
                System.out.println("ERROR: " + ex.getMessage());
            }
            // 6. Set the status to a legal value
            order.setAttribute("OrderStatusCode", "PENDING");
            // 7. Show the value of the status was updated successfully
            System.out.println("Current status is: " +
                order.getAttribute("OrderStatusCode"));
            // 8. Show the current value of the customer for this order
            System.out.println("Customer: " + order.getAttribute("CustomerId"));
            // 9. Reassign the order to customer # 113
```

```

        order.setAttribute("CustomerId",113); // Luis Popp
        // 10. Show the value of the reference information now
        System.out.println("Customer: "+order.getAttribute("CustomerId"));
        // 11. Rollback the transaction
        am.getTransaction().rollback();
        System.out.println("Transaction cancelled");
    }
    Configuration.releaseRootApplicationModule(am,true);
}
}
}

```

Running this example produces the following output:

```

Current status is: Closed
ERROR: The status must be Open, Pending, or Closed
Current status is: Open
Assigned: bernst
Assigned: Luis Popp
Transaction cancelled

```

6.4.9 How to Create a New Order

Example 6–9 performs the following basic steps:

1. Finds the Orders view object instance.
2. Creates a new row and inserts it into the row set.
3. Shows the effect of entity object-related defaulting for CreatedBy attribute.
4. Sets values of some of the required attributes in the new row.
5. Commits the transaction.
6. Retrieves and displays the trigger-assigned order ID.

Example 6–9 Creating a New Service Request

```

package oracle.fodemo.storefront.client;
import oracle.jbo.ApplicationModule;
import oracle.jbo.Row;
import oracle.jbo.ViewObject;
import oracle.jbo.client.Configuration;
import oracle.jbo.domain.DBSequence;
import oracle.jbo.domain.Date;
import oracle.jbo.domain.Timestamp;

public class TestCreateOrder {
    public static void main(String[] args) throws Throwable {
        String amDef = "oracle.fodemo.storefront.store.service.StoreServiceAM";
        String config = "StoreServiceAMLocal";
        ApplicationModule am =
            Configuration.createRootApplicationModule(amDef, config);
        // 1. Find the Orders view object instance.
        ViewObject orders = am.findViewObject("Orders");
        // 2. Create a new row and insert it into the row set
        Row newOrder = orders.createRow();
        orders.insertRow(newOrder);
        // 3. Show the entity object-related defaulting for CreatedBy attribute
        System.out.println("CreatedBy defaults to: " +
                           newOrder.getAttribute("CreatedBy"));
        // 4. Set values for some of the required attributes
        Date now = new Date(new Timestamp(System.currentTimeMillis()));
    }
}

```

```
        newOrder.setAttribute("OrderDate", now);
        newOrder.setAttribute("OrderStatusCode", "PENDING");
        newOrder.setAttribute("OrderTotal", 500);
        newOrder.setAttribute("CustomerId", 110);
        newOrder.setAttribute("ShipToAddressId", 2);
        newOrder.setAttribute("ShippingOptionId", 2);
        newOrder.setAttribute("FreeShippingFlag", "N");
        newOrder.setAttribute("GiftwrapFlag", "N");
        // 5. Commit the transaction
        am.getTransaction().commit();
        // 6. Retrieve and display the trigger-assigned service request id
        DBSequence id = (DBSequence)newOrder.getAttribute("OrderId");
        System.out.println("Thanks, reference number is " +
                           id.getSequenceNumber());
        Configuration.releaseRootApplicationModule(am, true);
    }
}
```

Running this example produces the following results:

```
CreatedBy defaults to: Luis Popp
Thanks, reference number is 200
```

6.4.10 How to Retrieve the Row Key Identifying a Row

Example 6-10 performs the following basic steps:

1. Finds the Orders view object.
2. Constructs a key to find order number 1011.
3. Finds the Orders row with this key.
4. Displays the key of the Orders row.
5. Accesses the row set of Orders using the OrderItems view link accessor attribute.
6. Iterates over the OrderItems row.
7. Displays the key of each OrderItems row.

Example 6-10 Retrieving the Row Key Identifying a Row

```
package oracle.fodemo.storefront.client;
import oracle.jbo.ApplicationModule;
import oracle.jbo.Key;
import oracle.jbo.Row;
import oracle.jbo.RowSet;
import oracle.jbo.ViewObject;
import oracle.jbo.client.Configuration;

public class TestFindAndShowKeys {
    public static void main(String[] args) {
        String amDef = "oracle.fodemo.storefront.store.service.StoreServiceAM";
        String config = "StoreServiceAMLocal";
        ApplicationModule am =
            Configuration.createRootApplicationModule(amDef, config);
        // 1. Find the Orders view object instance
        ViewObject vo = am.findViewObject("Orders");
        // 2. Construct a key to find order number 1011
        Key orderKey = new Key(new Object[] { 1011 });
        // 3. Find the Orders row with the key
```

```

Row[] ordersFound = vo.findByKey(orderKey, 1);
if (ordersFound != null && ordersFound.length > 0) {
    Row order = ordersFound[0];
    // 4. Displays the key of the Orders row
    showKeyFor(order);
    // 5. Accesses row set of Orders using OrderItems view link accessor
    RowSet items = (RowSet)order.getAttribute("OrderItems");
    // 6. Iterates over the OrderItems row
    while (items.hasNext()) {
        Row itemRow = items.next();
        // 4. Displays the key of each OrderItems row
        showKeyFor(itemRow);
    }
}
Configuration.releaseRootApplicationModule(am, true);
}

private static void showKeyFor(Row r) {
    // get the key for the row passed in
    Key k = r.getKey();
    // format the key as "(val1,val2)"
    String keyAttrs = formatKeyAttributeNamesAndValues(k);
    // get the serialized string format of the key, too
    String keyStringFmt = r.getKey().toStringFormat(false);
    System.out.println("Key " + keyAttrs + " has string format " +
        keyStringFmt);
}
// Build up "(val1,val2)" string for key attributes

private static String formatKeyAttributeNamesAndValues(Key k) {
    StringBuffer sb = new StringBuffer("(");
    int attrsInKey = k.getAttributeCount();
    for (int i = 0; i < attrsInKey; i++) {
        if (i > 0)
            sb.append(",");
        sb.append(k.getAttributeValues()[i]);
    }
    sb.append(")");
    return sb.toString();
}
}

```

Running the example produces the following results. Notice that the serialized string format of a key is a hexadecimal number that includes information in a single string that represents all the attributes in a key.

```

Key (1011) has string format 00010000003313031
Key (1011,1) has string format 000200000003C2020200000002C102
Key (1011,2) has string format 000200000003C2020200000002C103

```

Defining Validation and Business Rules Declaratively

This chapter explains the key entity object features for implementing the most common kinds of validation rules.

This chapter includes the following sections:

- [Section 7.1, "Introduction to Declarative Validation"](#)
- [Section 7.2, "Understanding the Validation Cycle"](#)
- [Section 7.3, "Adding Validation Rules to Entity Objects and Attributes"](#)
- [Section 7.4, "Using the Built-in Declarative Validation Rules"](#)
- [Section 7.5, "Using Groovy Expressions For Validation and Business Rules"](#)
- [Section 7.6, "Triggering Validation Execution"](#)
- [Section 7.7, "Creating Validation Error Messages"](#)
- [Section 7.8, "Setting the Severity Level for Validation Exceptions"](#)
- [Section 7.9, "Bulk Validation in SQL"](#)

7.1 Introduction to Declarative Validation

The easiest way to create and manage validation rules is through *declarative validation rules*. Declarative validation rules are defined using the overview editor, and once created, are stored in the entity object's XML file. Declarative validation is different from programmatic validation (covered in [Chapter 8, "Implementing Validation and Business Rules Programmatically"](#)), which is stored in an entity object's Java file.

Oracle ADF provides built-in declarative validation rules that satisfy many of your business needs. If you have custom validation rules you want to reuse, you can code them and add them to the IDE, so that the rules are available directly from JDeveloper. Custom validation rules are an advanced topic and covered in [Section 34.10, "Implementing Custom Validation Rules"](#). You can also base validation on a Groovy expression, as described in [Section 7.5, "Using Groovy Expressions For Validation and Business Rules"](#).

When you add a validation rule, you supply an appropriate error message and can later translate it easily into other languages if needed. You can also define how validation is triggered and set the severity level.

One benefit of using declarative validation (versus writing your own validation) is that the validation framework takes care of the complexities of batching validation

exceptions, which frees you to concentrate on your application's specific validation rule logic.

Note: It is possible to go beyond the declarative behavior to implement more complex validation rules for your business domain layer when needed. [Section 8.2, "Using Method Validators"](#) explains how to use the Method validator to invoke custom validation code and [Section 34.10, "Implementing Custom Validation Rules"](#) details how to extend the basic set of declarative rules with custom rules of your own.

7.1.1 When to Use Business-Layer Validation or Model-Layer Validation

In an ADF Business Components application, most of your validation code is defined in your entity objects. Encapsulating the business logic in these shared, reusable components ensures that your business information is validated consistently in every view object or client that accesses it, and it simplifies maintenance by centralizing where the validation is stored.

In the model layer, ADF Model validation rules can be set for the attributes of a collection. Many of the declarative validation features available for entity objects are also available at the model layer, should your application warrant the use of model-layer validation in addition to business-layer validation.

Unless you use data controls other than the ADF Business Components application module data control, you do not need to use model-layer validation. Consider defining all or most of your validation rules in the centralized, reusable, and easier to maintain entity objects of your business layer.

7.2 Understanding the Validation Cycle

Each entity row tracks whether or not its data is valid. When an existing entity row is retrieved from the database, the entity is assumed to be valid. When the first persistent attribute of an existing entity row is modified, or when a new entity row is created, the entity is marked invalid.

When an entity is in an invalid state, the declarative validation you have configured and the programmatic validation rules you have implemented are evaluated again before the entity can be considered valid again. You can determine whether a given entity row is valid at runtime by calling the `isValid()` method on it.

Note: Because attributes can (by default) be left blank, validations are not triggered if the attribute contains no value. For example, if a user creates a new entity row and does not enter a value for a given attribute, the validation on that attribute is not run. To force the validation to execute in this situation, set the Mandatory flag on the attribute.

7.2.1 Types of Entity Object Validation Rules

Entity object validation rules fall into two basic categories: *attribute-level* and *entity-level*.

7.2.1.1 Attribute-Level Validation Rules

Attribute-level validation rules are triggered for a particular entity object attribute when either the end user or the program code attempts to modify the attribute's value. Since you cannot determine the order in which attributes will be set, attribute-level validation rules should be used only when the success or failure of the rule depends exclusively on the candidate value of that single attribute.

The following examples are attribute-level validations:

- The value of the `OrderDate` of an order should not be a date in the past.
- The `ProductId` attribute of a product should represent an existing product.

7.2.1.2 Entity-Level Validation Rules

All other kinds of validation rules are entity-level validation rules. These are rules whose implementation requires considering two or more entity attributes, or possibly composed children entity rows, in order to determine the success or failure of the rule.

The following examples are entity-level validations:

- The value of the `OrderShippedDate` should be a date that comes after the `OrderDate`.
- The `ProductId` attribute of an order should represent an existing product.

Entity-level validation rules are triggered by calling the `validate()` method on a Row. This occurs when:

- You call the method explicitly on the entity object
- You call the method explicitly on a view row with an entity row part that is invalid
- A view object's iterator calls the method on the current row in the view object before allowing the current row to change
- During transaction commit, processing validates an invalid entity (in the list of pending changes) before proceeding with posting the changes to the database

As part of transaction commit processing, entity-level validation rules can fire multiple times (up to a specified limit). For more information, see [Section 7.2.4, "Avoiding Infinite Validation Cycles"](#).

7.2.2 Understanding Commit Processing and Validation

Transaction commit processing happens in three basic phases:

1. Ensure that any invalid entity rows on the pending changes list are valid.
2. Post the pending changes to the database by performing appropriate DML operations.
3. Commit the transaction.

If you have business validation logic in your entity objects that executes queries or stored procedures that depend on seeing the posted changes in the SELECT statements they execute, they should be coded in the `beforeCommit()` method described in [Section 8.5.2, "How to Validate Conditions Related to All Entities of a Given Type"](#). This method fires after all DML statements have been applied so queries or stored procedures invoked from that method can "see" all of the pending changes that have been saved, but not yet committed.

Caution: don't use the transaction-level `postChanges()` method in web applications unless you can guarantee that the transaction will definitely be committed or rolled-back during the same HTTP request. This method exists to force the transaction to post unvalidated changes without committing them. Failure to heed this advice can lead to strange results in an environment where both application modules and database connections can be pooled and shared serially by multiple different clients.

7.2.3 Understanding the Impact of Composition on Validation Order

Because a composed child entity row is considered an integral part of its composing parent entity object, any change to composed child entity rows causes the parent entity to be marked invalid. For example, if a line item on an order were to change, the entire order would now be considered to be changed, or invalid.

Therefore, when the composing entity is validated, it causes any currently invalid composed children entities to be validated first. This behavior is recursive, drilling into deeper levels of invalid composed children if they exist.

7.2.4 Avoiding Infinite Validation Cycles

If your validation rules contain code that updates attributes of the current entity or other entities, then the act of validating the entity can cause that or other entities to become invalid. As part of the transaction commit processing phase that attempts to validate all invalid entities in the pending changes list, the transaction performs multiple passes (up to a specified limit) on the pending changes list in an attempt to reach a state where all pending entity rows are valid.

The maximum number of validation passes is specified by the transaction-level validation threshold setting. The default value of this setting is 10. You can increase the threshold count to greater than one if the entities involved contain the appropriate logic to validate themselves in the subsequent passes.

If after 10 passes, there are still invalid entities in the list, you will see the following exception:

```
JBO-28200: Validation threshold limit reached. Invalid Entities still in cache
```

This is a sign that you need to debug your validation rule code to avoid inadvertently invalidating entities in a cyclic fashion.

7.2.5 What Happens When Validations Fail

When an entity object's validation rules throw exceptions, the exceptions are bundled and returned to the client. If the validation failures are thrown by methods you've overridden to handle events during the transaction `postChanges` processing, then the validation failures cause the transaction to roll back any database `INSERT`, `UPDATE`, or `DELETE` statements that might have been performed already during the current `postChanges` cycle.

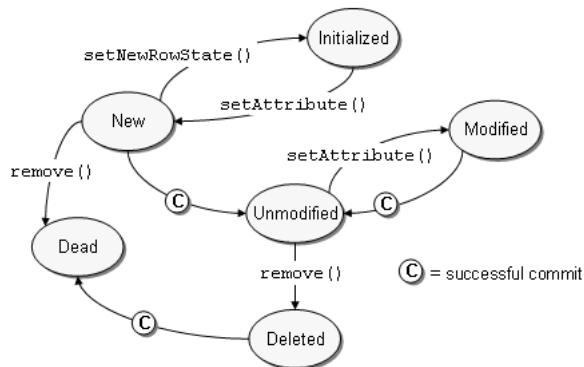
Note: The bundling of exceptions is the default behavior for ADF Model-based web applications, but not for tester or swing bindings. Additional configuration is required to bundle exceptions for tester or swing clients.

7.2.6 Understanding Entity Objects Row States

When an entity row is in memory, it has an entity state that reflects the logical state of the row. [Figure 7-1](#) illustrates the different entity row states and how an entity row can transition from one state to another. When an entity row is first created, its status is New. You can use the `setNewRowState()` method to mark the entity as being Initialized, which removes it from the transaction's list of pending changes until the user sets at least one of its attributes, at which time it returns to the New state. This allows you to create more than one initialized row and post only those that the user modifies.

The Unmodified state reflects an entity that has been retrieved from the database and has not yet been modified. It is also the state that a New or Modified entity transitions to after the transaction successfully commits. During the transaction in which it is pending to be deleted, an Unmodified entity row transitions to the Deleted state. Finally, if a row that was New and then was removed before the transaction commits, or Unmodified and then successfully deleted, the row transitions to the Dead state.

Figure 7-1 Diagram of Entity Row States and Transitions



You can use the `getEntityState()` method to access the current state of an entity row in your business logic code.

Note: If you use the `postChanges()` method to post pending changes without committing the transaction yet, the `getPostState()` method returns the entity's state from the point of view of it's being posted to the database or not. For example, a new entity row that has been inserted into the database due to your calling the `postChanges()` method programmatically — but which has not yet been committed — will have a different value for `getPostState()` and `getEntityState()`. The `getPostState()` method will reflect an "Unmodified" status since the new row has been posted, however the `getEntityState()` will still reflect that the entity is New in the current transaction.

7.3 Adding Validation Rules to Entity Objects and Attributes

The process for adding a validation rule to an entity object is similar for most of the validation rules, and is done using the Add Validation Rule dialog. You can open this dialog from the overview editor by clicking the **Add** icon on the Validators page.

7.3.1 How to Add a Validation Rule to an Entity or Attribute

To add a declarative validation rule to an entity object, use the Validators page of the overview editor.

To add a validation rule:

1. In the Application Navigator, double-click the desired entity object.
2. Click the **Validators** tab on the overview editor.
3. Select the object for which you want to add a validation rule, and then click the **Add** icon.
 - To add a validation rule at the entity object level, select **Entity**.
 - To add a validation rule for an attribute, expand **Attributes** and select the desired attribute.

When you add a new validation rule, the Add Validation Rule dialog appears.

4. Select the type of validation rule desired from the **Rule Type** dropdown list.
5. Use the dialog settings to configure the new rule.
The controls will change depending on the kind of validation rule you select.
6. You can optionally click the **Validation Execution** tab and enter criteria for the execution of the rule, such as dependent attributes and a precondition expression. For more information, see [Section 7.6, "Triggering Validation Execution"](#).

Note: For Key Exists and Method entity validators, you can also use the **Validation Execution** tab to specify the validation level.

7. Click the **Failure Handling** tab and enter or select the error message that will be shown to the user if the validation rule fails. For more information, see [Section 7.7, "Creating Validation Error Messages"](#).
8. Click **OK**.

7.3.2 How to View and Edit a Validation Rule On an Entity or Attribute

The Validators page of the overview editor for entity objects displays the validation rules for an entity and its attributes in a tree control. To see the validation rules that apply to the entity as a whole, expand in the **Entity** node. To see the validation rules that apply to an attribute, expand the **Attributes** node and then expand the attribute.

The validation rules that are shown on the Validators page of the overview editor include those that you have defined as well as database constraints, such as mandatory or precision. To open a validation rule for editing, double-click the rule or select the rule and click the **Edit** icon.

7.3.3 What Happens When You Add a Validation Rule

When you add a validation rule to an entity object, JDeveloper updates its XML component definition to include an entry describing what rule you've used and what rule properties you've entered. For example, if you add a range validation rule to the `DiscountAmount` attribute, this results in a `RangeValidationBean` entry in the XML file, as shown in [Appendix 7-1](#).

Example 7-1 Range Validation Bean

```

<Attribute
  Name="DiscountAmount"
  IsUpdateable="true"
  AttrLoad="Each"
  IsNotNull="true"
  ColumnName="DISCOUNT_AMOUNT"
  Type="oracle.jbo.domain.Number"
  ColumnType="NUMBER"
  SQLType="NUMERIC"
  TableName="DISCOUNTS_BASE"
  HistoryColumn="NonHistory">
  <DesignTime>
    <Attr Name="_DisplaySize" Value="22"/>
  </DesignTime>
  <validation:RangeValidationBean
    ResId="DiscountAmount_Rule_0"
    Severity="Warning"
    OnAttribute="DiscountAmount"
    OperandType="LITERAL"
    Inverse="false"
    MinValue="0"
    MaxValue="40"/>
</Attribute>
```

At runtime, the rule is enforced by the entity object based on this declarative information.

7.3.4 What You May Need to Know About Entity and Attribute Validation Rules

Declarative validation enforces both entity-level and attribute-level validation, depending on where you place the rules. Entity-level validation rules are enforced when a user tries to commit pending changes or navigates between rows.

Attribute-level validation rules are enforced when the user changes the value of the related attribute.

The Unique Key validator (described in [Section 7.4.1, "How to Ensure That Key Values Are Unique"](#)) can be used only at the entity level. Internally the Unique Key validator behaves like an attribute-level validator. This means that users see the validation error when they tab out of the key attribute for the key that the validator is validating. This is done because the internal cache of entities can never contain a duplicate, so it is not allowed for an attribute value to be set that would violate that. This check needs to be performed when the attribute value is being set because the cache consistency check is done during the setting of the attribute value.

Best Practice: If the validity of one attribute is dependent on one or more other attributes, enforce this rule using entity validation, not attribute validation. Examples of when you would want to do this include the following:

- You have a Compare validator that compares one attribute to another.
- You have an attribute with an expression validator that examines the value in another attribute to control branching in the expression to validate the attribute differently depending on the value in this other attribute.
- You make use of conditional execution, and your precondition expression involves an attribute other than the one that you are validating.

Entity object validators are triggered whenever the entity, as a whole, is dirty. To improve performance, you can indicate which attributes play a role in your rule and thus the rule should be triggered only if one or more of these attributes are dirty. For more information on triggering attributes, see, [Section 7.6, "Triggering Validation Execution"](#).

7.4 Using the Built-in Declarative Validation Rules

The built-in declarative validation rules can satisfy many, if not all, of your business needs. These rules are easy to implement because you don't write any code. You use the user-interface tools to choose the type of validation and how it is used.

Built-in declarative validation rules can be used to:

- Ensure that key values are unique (primary key or other unique keys)
- Determine the existence of a key value
- Make a comparison between an attribute and anything from a literal value to a SQL query
- Validate against a list of values that might be a literal list, a SQL query, or a view attribute
- Make sure that a value falls within a certain range, or that it is limited by a certain number of bytes or characters
- Validate using a regular expression or evaluate a Groovy expression
- Make sure that a value satisfies a relationship defined by an aggregate on a child entity available through an accessor
- Validate using a validation condition defined in a Java method on the entity

7.4.1 How to Ensure That Key Values Are Unique

The Unique Key validator ensures that primary key values for an entity object are always unique. The Unique Key validator can also be used for a non-primary-key attribute, as long as the attribute is defined as an alternate key. For information on how to define alternate keys, see [Section 4.10.15, "How to Define Alternate Key Values"](#).

Whenever any of the key attribute values change, this rule validates that the new key does not belong to any other entity object instance of this entity object class. (It is the business-logic tier equivalent of a unique constraint in the database.) If the key is

found in one of the entity objects, a `TooManyObjectsException` is thrown. The validation check is done both in the entity cache and in the database.

There is a slight possibility that unique key validation might not be sufficient to prevent duplicate rows in the database. It is possible for two application module sessions to simultaneously attempt to create records with the same key. To prevent this from happening, create a unique index in the database for any unique constraint that you want to enforce.

To ensure that a key value is unique:

1. In the Application Navigator, double-click the desired entity object.
2. On the Validators page of the overview editor, select the **Entity** folder, and click the **Add** icon.
3. In the Add Validation Rule dialog, in the **Rule Type** dropdown list, select **UniqueKey**.
4. In the **Keys** box, select the primary or alternate key.
5. You can optionally click the **Validation Execution** tab and enter criteria for the execution of the rule, such as dependent attributes and a precondition expression. For more information, see [Section 7.6, "Triggering Validation Execution"](#).

Best Practice: While it is possible to add a precondition for a Unique Key validator, it is not a best practice. If a Unique Key validator fails to fire, for whatever reason, the cache consistency check is still performed and an error will be returned. It is generally better to add the validator and a meaningful error message.

6. Click the **Failure Handling** tab and enter or select the error message that will be shown to the user if the validation rule fails. For more information, see [Section 7.7, "Creating Validation Error Messages"](#).
7. Click **OK**.

7.4.2 What Happens When You Use a Unique Key Validator

When you use a Unique Key validator, a `<UniqueKeyValidationBean>` tag is added to the entity object's XML file. [Example 7–2](#) shows the XML for a Unique Key validator.

Example 7–2 Unique Key Validator XML Code

```
<validation:UniqueKeyValidationBean
    Name="PersonEO_Rule_1"
    ResId="PersonEO_Rule_1"
    KeyName="AltKey" />
```

7.4.3 How to Validate Based on a Comparison

The Compare validator performs a logical comparison between an entity attribute and a value. When you add a Compare validator, you specify an operator and something to compare with. You can compare the following:

- Literal value

When you use a Compare validator with a literal value, the value in the attribute is compared against the specified literal value. When using this kind of comparison, it is important to consider data types and formats. The literal value must conform

to the format specified by the data type of the entity attribute to which you are applying the rule. In all cases, the type corresponds to the type mapping for the entity attribute.

For example, an attribute of column type DATE maps to the `oracle.jbo.domain.Date` class, which accepts dates and times in the same format accepted by `java.sql.Timestamp` and `java.sql.Date`. You can use format masks to ensure that the format of the value in the attribute matches that of the specified literal. For information about entity object attribute type mappings, see [Section 4.10.1, "How to Set Database and Java Data Types for an Entity Object Attribute"](#). For information about the expected format for a particular type, refer to the Javadoc for the type class.

- **Query result**

When you use this type of validator, the SQL query is executed each time the validator is executed. The validator retrieves the first row from the query result, and it uses the value of the first column in the query (of that first row) as the value to compare. Because this query cannot have any bind variables in it, this feature should be used only when selecting one column of one row of data that does not depend on the values in the current row.

- **View object attribute**

When you use this type of validator, the view object's SQL query is executed each time the validator is executed. The validator retrieves the first row from the query result, and it uses the value of the selected view object attribute from that row as the value to compare. Because you cannot associate values with the view object's named bind variables, those variables can only take on their default values.

Therefore this feature should be used only for selecting an attribute of one row of data that does not depend on the values in the current row.

- **View accessor attribute**

When defining the view accessor, you can assign row-specific values to the validation view object's bind variables.

- **Expression**

For information on the expression option, see [Section 7.5, "Using Groovy Expressions For Validation and Business Rules"](#).

- **Entity attribute**

The entity attribute option is available only for entity-level Compare validators.

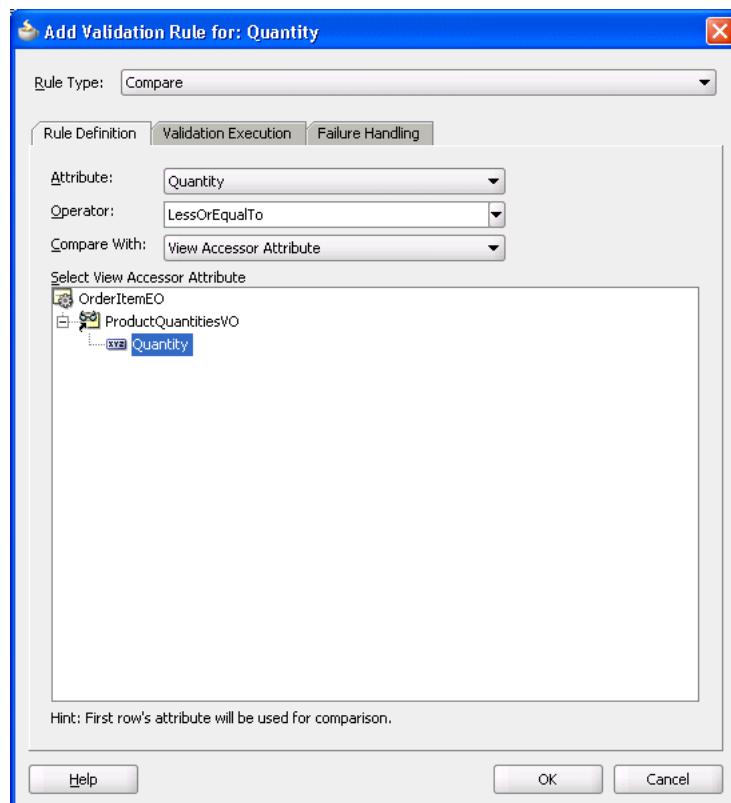
To validate based on a comparison:

1. In the Application Navigator, double-click the desired entity object.
2. On the Validators page of the overview editor, select where you want to add the validator.
 - To add an entity-level validator, select the **Entity** folder.
 - To add an attribute-level validator, expand the **Attributes** folder and select the appropriate attribute.
3. Click the **Add** icon.
4. In the Add Validation Rule dialog, in the **Rule Type** dropdown list, select **Compare**. Note that the subordinate fields change depending on your choices.
5. Select the appropriate operator.

6. Select an item in the **Compare With** list, and based on your selection provide the appropriate comparison value.
7. You can optionally click the **Validation Execution** tab and enter criteria for the execution of the rule, such as dependent attributes and a precondition expression. For more information, see [Section 7.6, "Triggering Validation Execution"](#).
8. Click the **Failure Handling** tab and enter or select the error message that will be shown to the user if the validation rule fails. For more information, see [Section 7.7, "Creating Validation Error Messages"](#).
9. Click **OK**.

[Figure 7-2](#) shows what the dialog looks like when you use a Compare validator with a view accessor attribute.

Figure 7-2 Compare Validator Using a View Object Attribute



7.4.4 What Happens When You Validate Based on a Comparison

When you use a Compare validator, a `<CompareValidationBean>` tag is added to an entity object's XML file. [Example 7-3](#) shows the XML code for the `Quantity` attribute in the `OrderItemEO` entity object.

Example 7-3 Compare Validator XML Code

```
<validation:CompareValidationBean
    Name="Quantity_Rule_0"
    ResId="Quantity_Rule_0"
    Severity="Warning"
    OnAttribute="Quantity"
    OperandType="VO_USAGE"
```

```
Inverse="false"
CompareType="LESSTHANEQUALTO"
ViewAccAttrName="Quantity"
ViewAccName="ProductQuantitiesVO" />
```

7.4.5 How to Validate Using a List of Values

The List validator compares an attribute against a list of values (LOV). When you add a List validator, you specify the type of list to choose from:

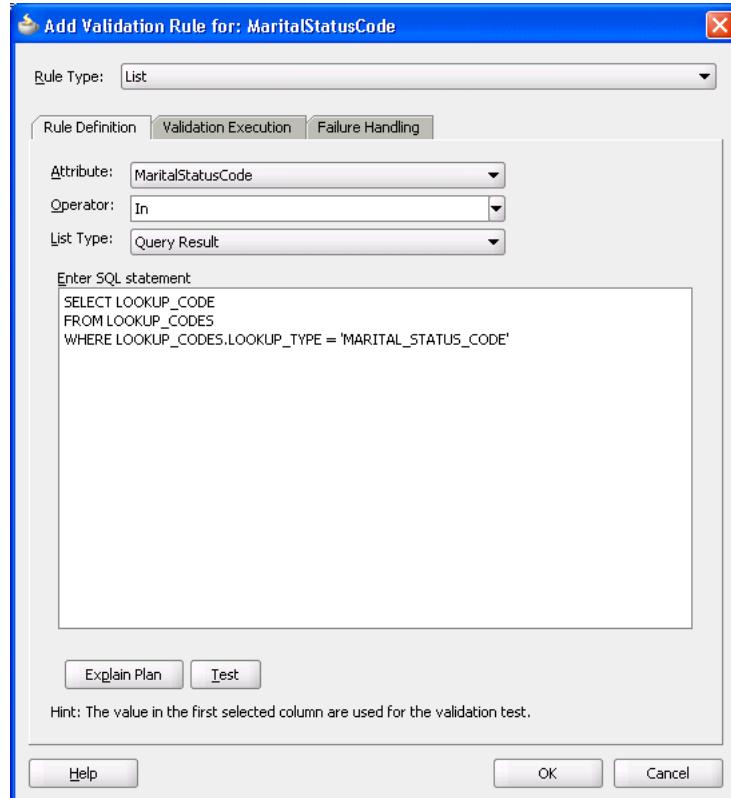
- Literal values - The validator ensures that the entity attribute is in (or not in, if specified) the list of values.
- Query result - The validator ensures that the entity attribute is in (or not in, if specified) the first column of the query's result set. The SQL query validator cannot use a bind variable, so it should be used only on a fixed, small list that you have to query from a table. All rows of the query are retrieved into memory.
- View object attribute - The validator ensures that the entity attribute is in (or not in, if specified) the view attribute. The View attribute validator cannot use a bind variable, so it should be used only on a fixed, small list that you have to query from a table. All rows of the query are retrieved into memory.
- View accessor attribute - The validator ensures that the entity attribute is in (or not in) the view accessor attribute. The view accessor is probably the most useful option, because it can take bind variables and after you've created the LOV on the user interface, a view accessor is required.

To validate using a list of values:

1. In the Application Navigator, double-click the desired entity object.
2. On the Validators page of the overview editor, select where you want to add the validator.
 - To add an entity-level validator, select the **Entity** folder.
 - To add an attribute-level validator, expand the **Attributes** folder and select the appropriate attribute.
3. Click the **Add** icon.
4. In the Add Validation Rule dialog, in the **Rule Type** dropdown list, select **List**.
5. In the **Attribute** list, choose the appropriate attribute.
6. In the **Operator** field, select **In** or **NotIn**, depending on whether you want an inclusive list or exclusive.
7. In the **List Type** field, select the appropriate type of list.
8. Depending on the type of list you selected, you can either enter a list of values (each value on a new line) or an SQL query, or select a view object attribute or view accessor attribute.
9. You can optionally click the **Validation Execution** tab and enter criteria for the execution of the rule, such as dependent attributes and a precondition expression. For more information, see [Section 7.6, "Triggering Validation Execution"](#).
10. Click the **Failure Handling** tab and enter or select the error message that will be shown to the user if the validation rule fails. For more information, see [Section 7.7, "Creating Validation Error Messages"](#).
11. Click **OK**.

Figure 7-3 shows what the dialog looks like when you use a List validator with a query.

Figure 7-3 Using a Query in a List Validator



7.4.6 What Happens When You Validate Using a List of Values

When you validate using a list of values, a `<ListValidationBean>` tag is added to an entity object's XML file. Example 7-4 shows the `PersonEO.MaritalStatusCode` attribute, which uses a query for the List validator.

Example 7-4 List Validator XML Code

```
<validation>ListValidationBean
    Name="MaritalStatusCode_Rule_0"
    ResId="MaritalStatusCode_Rule_0"
    OnAttribute="MaritalStatusCode"
    OperandType="SQL"
    Inverse="false"
    ListValue="SELECT LOOKUP_CODE
        FROM LOOKUP_CODES
        WHERE LOOKUP_CODES.LOOKUP_TYPE = 'MARITAL_STATUS_CODE'"/>
```

7.4.7 What You May Need to Know About the List Validator

The List validator is designed for validating an attribute against a relatively small set of values. If you select the **Query Result** or **View Object Attribute** type of list validation, keep in mind that the validator retrieves all of the rows from the query before performing an in-memory scan to validate whether the attribute value in question matches an attribute in the list. The query performed by the validator's SQL

or view object query does not reference the value being validated in the WHERE clause of the query.

It is inefficient to use a validation rule when you need to determine whether a user-entered product code exists in a table of a large number of products. Instead, [Section 8.5, "Using View Objects for Validation"](#) explains the technique you can use to efficiently perform SQL-based validations by using a view object to perform a targeted validation query against the database. See also [Section 5.11.7.2, "Using Validators to Validate Attribute Values"](#).

Also, if the attribute you're comparing to is a key, the Key Exists validator is more efficient than validating a list of values; and if these choices need to be translatable, you should use a static view object instead of the literal choice.

7.4.8 How to Make Sure a Value Falls Within a Certain Range

The Range validator performs a logical comparison between an entity attribute and a range of values. When you add a Range validator, you specify minimum and maximum literal values. The Range validator verifies that the value of the entity attribute falls within the range (or outside the range, if specified).

If you need to dynamically calculate the minimum and maximum values, or need to reference other attributes on the entity, use the Script Expression validator and provide a Groovy expression. See [Section 4.10.7, "How to Define a Default Value Using a Groovy Expression"](#).

To validate within a certain range:

1. In the Application Navigator, double-click the desired entity object.
2. On the Validators page of the overview editor, select where you want to add the validator.
 - To add an entity-level validator, select the **Entity** folder.
 - To add an attribute-level validator, expand the **Attributes** folder and select the appropriate attribute.
3. Click the **Add** icon.
4. In the Add Validation Rule dialog, in the **Rule Type** dropdown list, select **Range**.
5. In the **Attribute** list, select the appropriate attribute.
6. In the **Operator** field, select **Between** or **NotBetween**.
7. In the **Minimum** and **Maximum** fields, enter appropriate values.
8. You can optionally click the **Validation Execution** tab and enter criteria for the execution of the rule, such as dependent attributes and a precondition expression. For more information, see [Section 7.6, "Triggering Validation Execution"](#).
9. Click the **Failure Handling** tab and enter or select the error message that will be shown to the user if the validation rule fails. For more information, see [Section 7.7, "Creating Validation Error Messages"](#).
10. Click **OK**.

7.4.9 What Happens When You Use a Range Validator

When you validate against a range, a `<RangeValidationBean>` tag is added to the entity object's XML file. [Example 7-5](#) shows the `PersonEO.CreditLimit` attribute with a minimum credit limit of zero and a maximum of 10,000.

Example 7–5 Range Validator XML Code

```
<validation:RangeValidationBean
    Name="CreditLimit_Rule_0"
    ResId="CreditLimit_Rule_0"
    OnAttribute="CreditLimit"
    OperandType="LITERAL"
    Inverse="false"
    MinValue="0"
    MaxValue="10000" />
```

7.4.10 How to Validate Against a Number of Bytes or Characters

The Length validator validates whether the string length (in characters or bytes) of an attribute's value is less than, equal to, or greater than a specified number, or whether it lies between a pair of numbers.

To validate against a number of bytes or characters:

1. In the Application Navigator, double-click the desired entity object.
2. On the Validators page of the overview editor, select where you want to add the validator.
 - To add an entity-level validator, select the **Entity** folder.
 - To add an attribute-level validator, expand the **Attributes** folder and select the appropriate attribute.
3. Click the **Add** icon.
4. In the Add Validation Rule dialog, in the **Rule Type** dropdown list, select **Length**.
5. In the **Attribute** list, select the appropriate attribute.
6. In the **Operator** field, select how to evaluate the value.
7. In the **Comparison Type** field, select **Byte** or **Character** and enter a length.
8. You can optionally click the **Validation Execution** tab and enter criteria for the execution of the rule, such as dependent attributes and a precondition expression. For more information, see [Section 7.6, "Triggering Validation Execution"](#).
9. Click the **Failure Handling** tab and enter or select the error message that will be shown to the user if the validation rule fails. For more information, see [Section 7.7, "Creating Validation Error Messages"](#).
10. Click **OK**.

7.4.11 What Happens When You Validate Against a Number of Bytes or Characters

When you validate using length, a `<LengthValidationBean>` tag is added to the entity object's XML file, as shown in [Example 7–6](#). For example, you might have a field where the user enters a password or PIN and the application wants to validate that it is at least 6 characters long, but not longer than 10. You would use the Length validator with the Between operator and set the minimum and maximum values accordingly.

Example 7–6 Validating the Length Between Two Values

```
<validation:LengthValidationBean
    OnAttribute="pin"
    CompareType="BETWEEN"
    DataType="CHARACTER"
```

```
MinValue="6"
MaxValue="10"
Inverse="false"/>
```

7.4.12 How to Validate Using a Regular Expression

The Regular Expression validator compares attribute values against a mask specified by a Java regular expression.

If you want to create expressions that can be personalized in metadata, you can use the Script Expression validator. For more information, see [Section 7.5, "Using Groovy Expressions For Validation and Business Rules"](#).

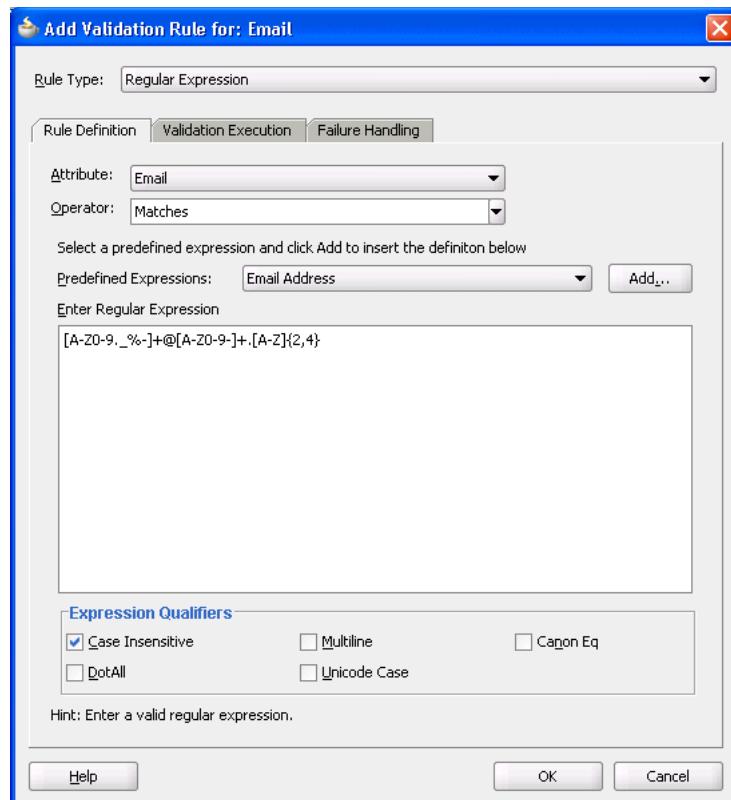
To validate using a regular expression

1. In the Application Navigator, double-click the desired entity object.
2. On the Validators page of the overview editor, select where you want to add the validator.
 - To add an entity-level validator, select the **Entity** folder.
 - To add an attribute-level validator, expand the **Attributes** folder and select the appropriate attribute.
3. Click the **Add** icon.
4. In the Add Validation Rule dialog, in the **Rule Type** dropdown list, select **Regular Expression**.
5. In the **Operator** field, you can select **Matches** or **Not Matches**.
6. To use a predefined expression (if available), you can select one from the dropdown list and click **Use Pattern**. Otherwise, write your own regular expression in the field provided.

Note: You can add your own expressions to the list of predefined expressions. To add a predefined expression, add an entry in the `PredefinedRegExp.properties` file in the BC4J subdirectory of the JDeveloper system directory (for example, `C:\Documents and Settings\username\Application Data\JDeveloper\system\oracle.BC4J.version#\PredefinedRegExp.properties`).

7. You can optionally click the **Validation Execution** tab and enter criteria for the execution of the rule, such as dependent attributes and a precondition expression. For more information, see [Section 7.6, "Triggering Validation Execution"](#).
8. Click the **Failure Handling** tab and enter or select the error message that will be shown to the user if the validation rule fails. For more information, see [Section 7.7, "Creating Validation Error Messages"](#).
9. Click **OK**.

[Figure 7–4](#) shows what the dialog looks like when you select a Regular Expression validator and validate that the `Email` attribute matches a predefined **Email Address** expression.

Figure 7–4 Regular Expression Validator Matching Email Address

7.4.13 What Happens When You Validate Using a Regular Expression

When you validate using a regular expression, a `<RegExpValidationBean>` tag is added to the entity object's XML file. [Example 7–7](#) shows an `Email` attribute that must match a regular expression.

Example 7–7 Regular Expression Validator XML Code

```
<validation:RegExpValidationBean
    Name="Persons_Rule_3"
    OnAttribute="Email"
    Pattern="[A-Za-z0-9_.!#$%&'*+-/=?^`{|}~]+@[A-Z0-9-].[A-Z]{2,4}"
    Flags="CaseInsensitive"
    Inverse="false"/>
```

7.4.14 How to Use the Average, Count, or Sum to Validate a Collection

You can use collection validation on the average, count, sum, min, or max of a collection. This validator is available only at the entity level. It is useful for validating the aggregate calculation over a collection of associated entities by way of an entity accessor to a child entity (on the many end of the association). You must select the association accessor to define the Collection validator.

To validate using the average, count, or sum:

1. In the Application Navigator, double-click the desired entity object.
2. On the Validators page of the overview editor, select the **Entity** folder and click the **Add** icon.

3. In the Add Validation Rule dialog, in the **Rule Type** dropdown list, select **Collection**.
4. In the **Operation** field, specify the operation (sum, average, count, min, or max) to perform on the collection for comparison.
5. Select the appropriate accessor and attribute for the validation.
6. Specify the operator and the comparison type and value.
7. You can optionally click the **Validation Execution** tab and enter criteria for the execution of the rule, such as dependent attributes and a precondition expression. For more information, see [Section 7.6, "Triggering Validation Execution"](#).
8. Click the **Failure Handling** tab and enter or select the error message that will be shown to the user if the validation rule fails. For more information, see [Section 7.7, "Creating Validation Error Messages"](#).
9. Click **OK**.

7.4.15 What Happens When You Use Collection Validation

When you validate using a Collection validator, a `<CollectionValidationBean>` tag is added to the entity object's XML file, as in [Example 7-8](#).

Example 7-8 Collection Validator XML Code

```
<validation:CollectionValidationBean
    Name="OrderEO_Rule_0"
    OnAttribute="OrderTotal"
    OperandType="LITERAL"
    Inverse="false"
    CompareType="LESSTHAN"
    CompareValue="5"
    Operation="sum" />
```

7.4.16 How to Determine Whether a Key Exists

The Key Exists validator is used to determine whether a key value (primary, foreign, or alternate key) exists.

There are a couple of benefits to using the Key Exists validator:

- The Key Exists validator has better performance because it first checks the cache and only goes to the database if necessary.
- Since the Key Exists validator uses the cache, it will find a key value that has been added in the current transaction, but not yet committed to the database. For example, you add a new Department and then you want to link an Employee to that new department.

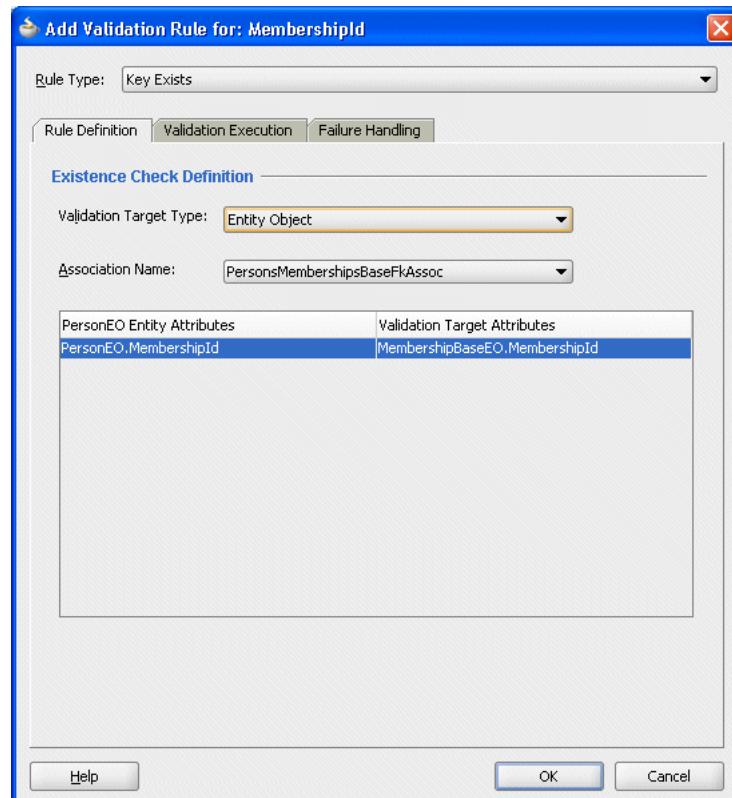
To determine whether a value exists:

1. In the Application Navigator, double-click the desired entity object.
2. On the Validators page of the overview editor, select where you want to add the validator.
 - To add an entity-level validator, select the **Entity** folder.
 - To add an attribute-level validator, expand the **Attributes** folder and select the appropriate attribute.
3. Click the **Add** icon.

4. In the Add Validation Rule dialog, select **Key Exists** from the **Rule Type** list.
5. Select the type of validation target (**Entity Object**, **View Object**, or **View Accessor**).
If you want the Key Exists validator to be used for all view objects that use this entity attribute, select **Entity Object**.
6. Depending on the validation target, you can choose either an association or a key value.
If you are searching for an attribute that does not exist in the **Validation Target Attributes** list, it is probably not defined as a key value. To create alternate keys, see [Section 4.10.15, "How to Define Alternate Key Values"](#).
7. You can optionally click the **Validation Execution** tab and enter criteria for the execution of the rule, such as dependent attributes and the validation level (entity or transaction). For more information, see [Section 7.6, "Triggering Validation Execution"](#).
8. Click the **Failure Handling** tab and enter or select the error message that will be shown to the user if the validation rule fails. For more information, see [Section 7.7, "Creating Validation Error Messages"](#).
9. Click **OK**.

[Figure 7-5](#) shows a Key Exists validator that validates whether the `MembershipId` entered in the `PersonEO` entity object exists in the `MembershipBaseEO` entity object.

Figure 7-5 Key Exists Validator on an Entity Attribute



7.4.17 What Happens When You Use a Key Exists Validator

When you use a Key Exists validator, an <ExistsValidationBean> tag is created in the XML file for the entity object, as in [Example 7-9](#).

Example 7-9 Using the Key Exists Validator With an Association

```
<validation:ExistsValidationBean
    Name="MembershipId_Rule_0"
    ResId="MembershipId_Rule_0"
    OperandType="EO"
    AssocName=
    "oracle.fodemo.storefront.entities.associations.PersonsMembershipsBaseFkAssoc" />
```

7.4.18 What You May Need to Know About Declarative Validators and View Accessors

When using declarative validators you must consider how your validation will interact with expected input. The combination of declarative validators and view accessors provides a simple yet powerful alternative to coding. But, as powerful as the combination is, you still need to consider how data composition can impact performance.

Consider a scenario where you have the following:

- A ServiceRequestEO entity object with Product and RequestType attributes, and a view accessor that allows it to access the RequestTypeVO view object
- A RequestTypeVO view object with a query specifying the Product attribute as a bind parameter

The valid list of RequestTypes varies by Product. So, to validate the RequestType attribute, you use a List validator using the view accessor.

Now lets add a set of new service requests. For the first service request (row), the List validator binds the value of the Product attribute to the view accessor and executes it. For each subsequent service request the List validator compares the new value of the Product attribute to the currently bound value.

- If the value of Product matches, the current RowSet is retained.
- If the value of Product has changed, the new value is bound and the view accessor re-executed.

Now consider the expected composition of input data. For example, the same products could appear in the input multiple times. If you simply validate the data in the order received, you might end up with the following:

1. Dryer (initial query)
2. Washing Machine (re-execute view accessor)
3. Dish Washer (re-execute view accessor)
4. Washing Machine (re-execute view accessor)
5. Dryer (re-execute view accessor)

In this case, the validator will execute 5 queries to get 3 distinct RowSets. As an alternative, you can add an ORDER BY clause to the RequestTypeVO to sort it by Product. In this case, the validator would execute the query only once each for Washing Machine and Dryer.

1. Dish Washer (initial query)
2. Dryer (re-execute view accessor)

3. Dryer
4. Washing Machine (re-execute view accessor)
5. Washing Machine

A small difference on a data set this size, but multiplied over larger data sets and many users this could easily become an issue. An ORDER BY clause is not a solution to every issue, but this example illustrates how data composition can impact performance.

7.5 Using Groovy Expressions For Validation and Business Rules

Groovy expressions are Java-like scripting code stored in the XML definition of an entity object. Because Groovy expressions are stored in XML, you can change the expression values even if you don't have access to the entity object's Java file. You can even change or specify values at runtime.

For more information about using Groovy script in your entity object business logic, see [Section 3.6, "Overview of Groovy Support"](#).

7.5.1 How to Reference Entity Object Methods in Groovy Validation Expressions

You can call methods on the current entity instance using the source property of the current object. The source property allows you to access to the entity instance being validated.

If the method is a non-boolean type and the method name is `getXyzAbc()` with no arguments, then you access its value as if it were a property named `XyzAbc`. For a boolean-valued property, the same holds true but the JavaBean naming pattern for the getter method changes to recognize `isXyzAbc()` instead of `getXyzAbc()`. If the method on your entity object does not match the JavaBean getter method naming pattern, or if it takes one or more arguments, then you must call it like a method using its complete name.

For example, say you have an entity object with the four methods shown in [Example 7-10](#).

Example 7-10 Sample Entity Object Methods

```
public boolean isNewRow() {
    System.out.println("## isNewRow() accessed ##");
    return true;
}

public boolean isNewRow(int n) {
    System.out.println("## isNewRow(int n) accessed ##");
    return true;
}

public boolean testWhetherRowIsNew() {
    System.out.println("## testWhetherRowIsNew() accessed ##");
    return true;
}

public boolean testWhetherRowIsNew(int n) {
    System.out.println("## testWhetherRowIsNew(int n) accessed ##");
    return true;
}
```

Then the following Groovy validation condition would trigger them all, one of them being triggered twice, as shown in [Example 7-11](#).

Example 7-11 Groovy Script Calling Sample Methods

```
newRow && source.newRow && source.is newRow(5) && source.testWhetherRowIsNew() &&  
source.testWhetherRowIsNew(5)
```

By running this example and forcing entity validation to occur, you would see the diagnostic output shown in [Example 7-12](#) in the log window:

Example 7-12 Output From Sample Groovy Script

```
## is newRow() accessed ##  
## is newRow() accessed ##  
## is newRow(int n) accessed ##  
## testWhetherRowIsNew() accessed ##  
## testWhetherRowIsNew(int n) accessed ##
```

Notice the slightly different syntax for the reference to a method whose name matches the JavaBeans property getter method naming pattern. Both `newRow` and `source.newRow` work to access the boolean-valued, JavaBeans getter-style method that has no arguments. But because the `testWhetherRowIsNew` method does not match the JavaBeans getter method naming pattern, and the second `isRowNew` method takes an argument, then you must call them like methods using their complete name.

7.5.2 How to Validate Using a True/False Expression

You can use a Groovy expression to return a true/false statement. The Script Expression validator *requires* that the expression either return `true` or `false`, or that it calls the `adf.error.raise/warn()` method. A common use of this feature would be to validate an attribute value, for example, to make sure that an account number is valid.

To validate using a true/false expression:

1. In the Application Navigator, double-click the desired entity object.
2. On the Validators page of the overview editor, select where you want to add the validator.
 - To add an entity-level validator, select the **Entity** folder.
 - To add an attribute-level validator, expand the **Attributes** folder and select the appropriate attribute.
3. Click the **Add** icon.
4. In the Add Validation Rule dialog, in the **Rule Type** dropdown list, select **Script Expression**.
5. Enter a validation expression in the field provided.
6. You can optionally click the **Validation Execution** tab and enter criteria for the execution of the rule, such as dependent attributes and a precondition expression. For more information, see [Section 7.6, "Triggering Validation Execution"](#).
7. Click the **Failure Handling** tab and enter or select the error message that will be shown to the user if the validation rule fails. For more information, see [Section 7.7, "Creating Validation Error Messages"](#).

8. Click OK.

The sample code in [Example 7-13](#) comes from the PaymentOptionEO entity object. The code validates account numbers based on the Luhn algorithm, a checksum formula in widespread use.

Example 7-13 Validating an Account Number Using an Expression

```
<validation:ExpressionValidationBean
    Name="AccountNumber_Rule_0"
    OperandType="EXPR"
    Inverse="false">
    <MsgIds>
        <Item
            Value="PaymentOption_AccountNumber" />
    </MsgIds>
    <TransientExpression>
        <![CDATA[
            String acctnumber = newValue;
            sumofdigits = 0;
            digit = 0;
            addend = 0;
            timesTwo = false;
            range = acctnumber.length()-1..0
            range.each {i ->
                digit = Integer.parseInt (acctnumber.substring (i, i + 1));
                if (timesTwo) {
                    addend = digit * 2;
                    if (addend > 9) {
                        addend -= 9;
                    }
                }
                else {
                    addend = digit;
                }
                sumofdigits += addend;
                timesTwo = !timesTwo;
            }
            modulus = sumofdigits % 10;
            return modulus == 0;
        ]]>
    </TransientExpression>
</ExpressionValidationBean>
```

7.5.3 What Happens When You Add a Groovy Expression

When you create a Groovy expression, it is saved in the entity object's XML component. [Example 7-14](#) shows the RegisteredDate attribute in the PersonEO.xml file. The Groovy expression is wrapped by a `<TransientExpression>` tag.

Example 7-14 XML Code for RegisteredDate Attribute on the PersonEO Entity Object

```
<Attribute
    Name="RegisteredDate"
    IsUpdateable="true"
    ColumnName="REGISTERED_DATE"
    Type="oracle.jbo.domain.Date"
    ColumnType="DATE"
```

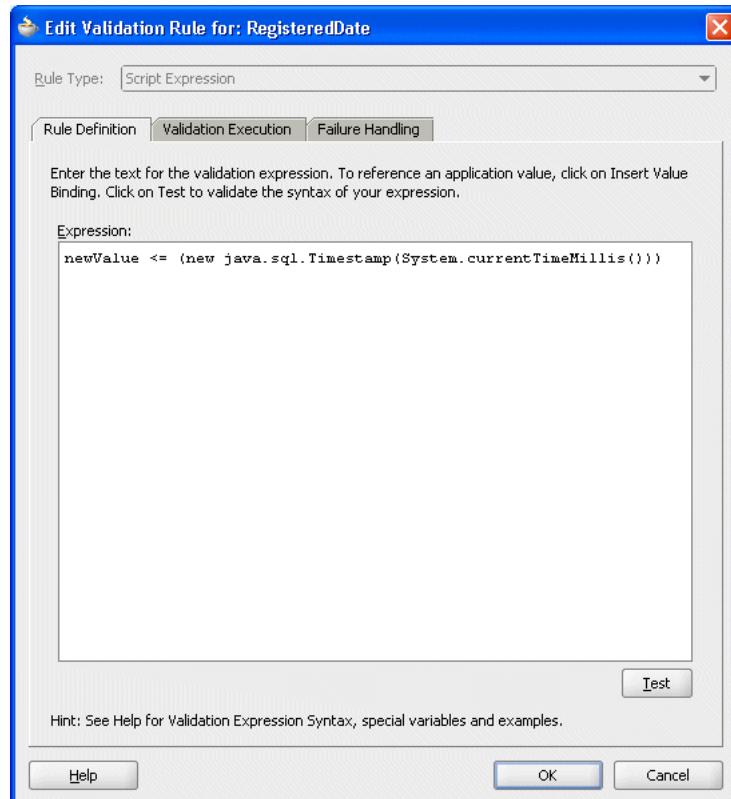
```

SQLType="DATE"
TableName="PERSONS">
<DesignTime>
<Attr Name="_DisplaySize" Value="7"/>
</DesignTime>
<validation:ExpressionValidationBean
  Name="RegisteredDate_Rule_0"
  OperandType="EXPR"
  Inverse="false">
<MsgIds>
  <Item
    Value="RegisteredDate_Rule_0"/>
</MsgIds>
<TransientExpression>
  <![CDATA[
newValue <= (new java.sql.Timestamp(System.currentTimeMillis()))
  ]]>
</TransientExpression>
</ExpressionValidationBean>
</Attribute>

```

This tag can take one of several forms. For some Groovy expressions, the `<TransientExpression>` tag is wrapped by an `<ExpressionValidationBean>` tag as well. [Figure 7–6](#) shows the validation expression in the Edit Validation Rule dialog.

Figure 7–6 Validation Expression for RegisteredDate Attribute on the PersonEO Entity Object



7.6 Triggering Validation Execution

JDeveloper allows you to select the attributes that trigger validation, so that validation execution happens only when one of the triggering attributes is dirty. In previous releases of JDeveloper, an entity-level validator would fire on an attribute whenever the entity as a whole was dirty. This feature is described in [Section 7.6.1, "How to Specify Which Attributes Fire Validation"](#).

JDeveloper also allows you to specify a precondition for the execution of a validator (as described in [Section 7.6.2, "How to Set Preconditions for Validation"](#)) and set transaction-level validation (described in [Section 7.6.3, "How to Set Transaction-Level Validation"](#)).

7.6.1 How to Specify Which Attributes Fire Validation

When defining a validator at the entity level, you have the option of selecting one or more attributes of the entity object that, when changed, trigger execution of the validator.

Note: When the validity of one attribute is dependent on the value in another attribute, the validation should be performed as entity validation, not attribute validation. You can set validation execution order on the entity level or attribute level.

If you do not specify one or more dependent attributes, the validator will fire whenever the entity is dirty. Firing execution only when required makes your application more performant.

To specify which attributes fire validation:

1. In the Application Navigator, double-click the desired entity object.
2. On the Validators page of the overview editor, select a validation rule and click the **Edit** icon.
3. In the Edit Validation Rule dialog, click the **Validation Execution** tab.
4. Select the attributes that will fire validation.
5. Click **OK**.

7.6.2 How to Set Preconditions for Validation

The **Validation Execution** tab (on the Add/Edit Validation Rule dialog) allows you to add a Groovy expression that serves as a precondition. If you enter an expression in the **Conditional Execution Expression** box, the validator is executed only if the condition evaluates True.

Best Practice: While it is possible to add a precondition for a Unique Key validator, it is not a best practice. If a Unique Key validator fails to fire, for whatever reason, the cache consistency check is still performed and an error will be returned. It is generally better to add the validator and a meaningful error message.

7.6.3 How to Set Transaction-Level Validation

Performing a validation during the transaction level (rather than entity level) means that the validation will be performed after all entity-level validation is performed. For

this reason, it may be useful if you want to ensure that a validator is performed at the end of the process.

In addition, the Key Exists validator is more performant with bulk transactions if it is run as a transaction level validator since it will be run only once for all entities in the transaction (of the same type), rather than once per entity. This will result in improved performance if the validator has to go to the database.

Note: Transaction-level validation is only applicable to Key Exists and Method entity validators.

To specify entity-level or transaction-level validation:

1. In the Application Navigator, double-click the desired entity object.
2. On the Validators page of the overview editor, select an entity-level validation rule and click the **Edit** icon.
3. In the Edit Validation Rule dialog, click the **Validation Execution** tab.
4. Select **Execute at Entity Level** or **Defer Execution to Transaction Level**.
5. Click **OK**.

7.6.4 What You May Need to Know About the Order of Validation Execution

You cannot control the order in which attributes are validated – they are always validated in the order they appear in the entity definition. You can order validations for a given attribute (or for the entity), but you cannot reorder the attributes themselves.

7.7 Creating Validation Error Messages

Validation error messages provide important information for the user: the message should convey what went wrong and how to fix it.

7.7.1 How to Create Validation Error Messages

When you create or edit a validation rule, enter text to help the user determine what caused the error.

To create validation error messages:

1. In the Application Navigator, double-click the desired entity object.
2. On the Validators page of the overview editor, select a validation rule and click the **Edit** icon.
3. In the Edit Validation Rule dialog, click the **Failure Handling** tab.
4. In the **Message Text** field, enter your error message.

You can also define error messages in a message bundle file. To select a previously defined error message or to define a new one in a message bundle file, click the **Select Message** icon.

Note: The Script Expression validator allows you to enter more than one error message. This is useful if the validation script conditionally returns different error or warning messages. For more information, see [Section 7.7.3, "How to Conditionally Raise Error Messages Using Groovy"](#).

5. You can optionally include message tokens in the body of the message, and define them in the **Token Message Expressions** list.

[Figure 7-7](#) shows the failure message for a validation rule in the `PaymentOptionEO` entity object that contains message tokens. For more information on this feature, see [Section 7.7.4, "How to Embed a Groovy Expression in an Error Message"](#).

6. Click **OK**.

7.7.2 How to Localize Validation Messages

The error message is a translatable string and is managed in the same way as translatable UI control hints in an entity object message bundle class. To view the error message for the defined rule in the message bundle class, locate the `String` key in the message bundle that corresponds to `ResId` property in the XML component definition entry for the validator. For example, [Example 7-15](#) shows a message bundle where the `NAME_CANNOT_BEGIN_WITH_U` key appears with the error message for the default locale.

Example 7-15 Message Bundle Contains Validation Error Messages

```
package devguide.advanced.customerrors;
import java.util.ListResourceBundle;

public class CustomMessageBundle extends ListResourceBundle {
    private static final Object[][][] sMessageStrings = new String[][][] {
        // other strings here
        {"NAME_CANNOT_BEGIN_WITH_U", "The name cannot begin with the letter u!"},
        // other strings here
    };
    // etc.
}
```

7.7.3 How to Conditionally Raise Error Messages Using Groovy

You can use the `adf.error.raise()` and `adf.error.warn()` methods to conditionally raise one error message or another depending upon branching in the Groovy expression. For example, if an attribute value is `x`, then validate as follows, and if the validation fails, raise error messageA; whereas if the attribute value is `y`, then instead validate a different way and if validation fails, raise error messageB.

If the expression returns `false` (versus raising a specific error message using the `raise()` method), the validator calls the first error message associated with the validator.

The syntax of the `raise()` method takes one required parameter (the `msgId` to use from the message bundle), and optionally can take the `attrName` parameter. If you pass in the `AttrName`, the error is associated with that attribute even if the validation is assigned to the entity.

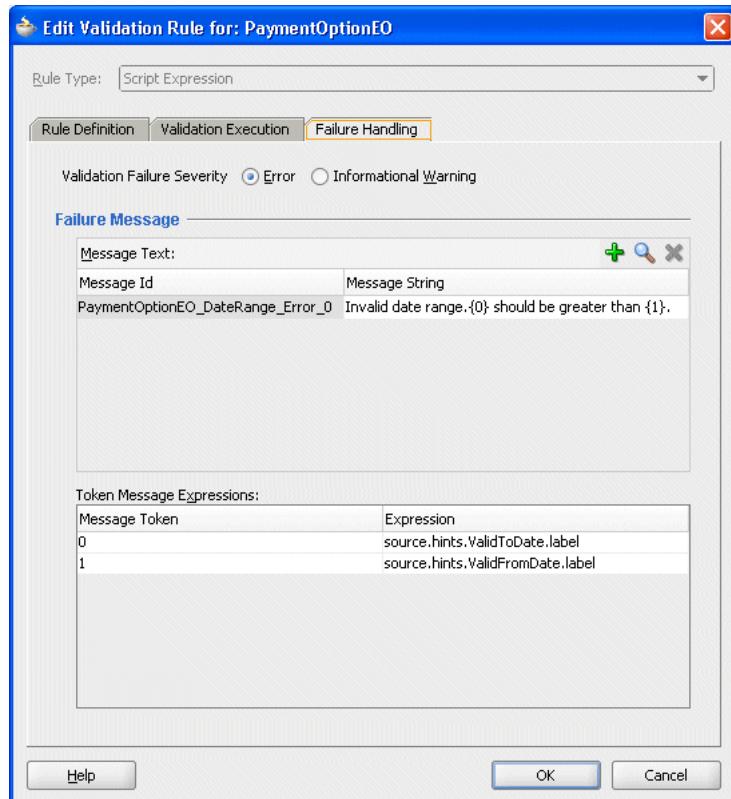
You can use either `adf.error.raise()` or `adf.error.warn()` methods, depending on whether you want to throw an exception, or whether you want processing to continue, as described in [Section 7.8, "Setting the Severity Level for Validation Exceptions"](#).

7.7.4 How to Embed a Groovy Expression in an Error Message

A validator's error message can contain embedded expressions that are resolved by the server at runtime. To access this feature, simply enter a named token delimited by curly braces (for example, `{2}` or `{errorParam}`) in the error message text where you want the result of the Groovy expression to appear.

After entering the token into the text of the error message (on the **Failure Handling** tab of the Edit Validation Rule dialog), the **Token Message Expressions** table at the bottom of the dialog displays a row that allows you to enter a Groovy expression for the token. [Figure 7–7](#) shows the failure message for a validation rule in the `PaymentOptionEO` entity object that contains message tokens.

Figure 7–7 Using Message Tokens in a Failure Message



7.8 Setting the Severity Level for Validation Exceptions

You can set the severity level for validation exceptions to two levels, Informational Warning and Error. If you set the severity level to Informational Warning, an error message will display, but processing will continue. If you set the validation level to Error, the user will not be able to proceed until you have fixed the error.

Under most circumstances you will use the Error level for validation exceptions, so this is the default setting. However, you might want to implement a Informational

Warning message if the user has a certain security clearance. For example, a store manager may want to be able to make changes that would surface as an error if a clerk tried to do the same thing.

To set the severity level for validation exceptions, use the **Failure Handling** tab of the Add Validation Rule dialog.

To set the severity level of a validation exception:

1. In the Application Navigator, double-click the desired entity object.
2. On the Validators page, select an existing validation rule and click the **Edit** icon, or click the **Add** icon to create a new rule.
3. In the Edit/Add Validation Rule dialog, click the **Failure Handling** tab and select the option for either **Error** or **Informational Warning**.
4. Click **OK**.

7.9 Bulk Validation in SQL

To improve the performance of batch-load applications, such as data synchronization programs, the ADF framework employs bulk validation for primary keys (including alternate keys) and foreign keys.

When the Key Exists validator is configured to defer validation until the transaction commits, or when the rows are being updated or inserted through the `processXXX` methods of the ADF business components service layer, the validation cache is preloaded. This behavior uses the normal row-by-row derivation and validation logic, but uses validation logic that checks a memory cache before making queries to the database. Performance is improved by preloading the memory cache using bulk SQL operations based on the inbound data.

8

Implementing Validation and Business Rules Programmatically

This chapter explains the key entity object events and features for implementing the most common kinds of business rules.

This chapter includes the following sections:

- [Section 8.1, "Introduction to Programmatic Business Rules"](#)
- [Section 8.2, "Using Method Validators"](#)
- [Section 8.3, "Assigning Programmatically Derived Attribute Values"](#)
- [Section 8.4, "Undoing Pending Changes to an Entity Using the Refresh Method"](#)
- [Section 8.5, "Using View Objects for Validation"](#)
- [Section 8.6, "Accessing Related Entity Rows Using Association Accessors"](#)
- [Section 8.7, "Referencing Information About the Authenticated User"](#)
- [Section 8.8, "Accessing Original Attribute Values"](#)
- [Section 8.9, "Storing Information About the Current User Session"](#)
- [Section 8.10, "Accessing the Current Date and Time"](#)
- [Section 8.11, "Sending Notifications Upon a Successful Commit"](#)
- [Section 8.12, "Conditionally Preventing an Entity Row from Being Removed"](#)
- [Section 8.13, "Determining Conditional Updatability for Attributes"](#)

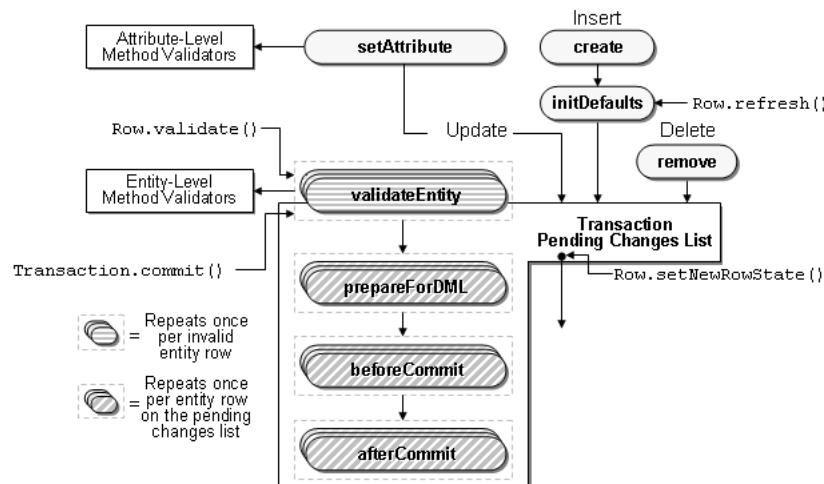
8.1 Introduction to Programmatic Business Rules

Complementing the built-in *declarative* validation features, entity objects and view objects have method validators and several events you can handle to *programmatically* implement encapsulated business logic using Java code. These concepts are illustrated in [Figure 8–1](#).

- Attribute-level method validators trigger validation code when an attribute value changes.
- Entity-level method validators trigger validation code when an entity row is validated.
- You can override the following key methods in a custom Java class for an entity:
 - `create()`, to assign default values when a row is created

- `initDefaultExpressionAttributes()`, to assign defaults either when a row is created or when a new row is refreshed
- `remove()`, to conditionally disallow deleting
- `isAttributeUpdateable()`, to make attributes conditionally updatable
- `setAttribute()`, to trigger attribute-level method validators
- `validateEntity()`, to trigger entity-level method validators
- `prepareForDML()`, to assign attribute values before an entity row is saved
- `beforeCommit()`, to enforce rules that must consider all entity rows of a given type
- `afterCommit()`, to send notifications about a change to an entity object's state

Figure 8–1 Key Entity Objects Features and Events for Programmatic Business Logic



Note: When coding programmatic business rules, it's important to have a firm grasp of the validation cycle. For more information, see [Section 7.2, "Understanding the Validation Cycle"](#).

8.2 Using Method Validators

Method validators are the primary way of supplementing declarative validation rules and Groovy-scripted expressions using your own Java code. Method validators trigger Java code that you write in your own validation methods at the appropriate time during the entity object validation cycle. There are many types of validation you can code with a method validator, either on an attribute or on an entity as a whole.

You can add any number of attribute-level or entity-level method validators, provided they each trigger a distinct method name in your code. All validation method names must begin with the word `validate`; however, following that rule you are free to name them in any way that most clearly identifies the functionality. For an attribute-level validator, the method must take a single argument of the same type as the entity attribute. For an entity-level validator, the method takes no arguments. The method must also be public, and must return a boolean value. Validation will fail if the method returns `false`.

Note: Although it is important to be aware of these rules, when you use JDeveloper to create method validators, JDeveloper creates the correct interface for the class.

At runtime, the Method validator passes an entity attribute to a method implemented in your entity object class.

In [Example 8–1](#), the method accepts strings that start with a capital letter and throws an exception on null values, empty strings, and strings that do not start with a capital letter.

Example 8–1 Method That Validates If the First Letter Is a Capital

```
public boolean validateIsCapped(String text)
{
    if (text != null &&
        text.length() != 0 &&
        text[0] >= 'A' &&
        text[0] <= 'Z')
    {
        return true;
    }
}
```

8.2.1 How to Create an Attribute-Level Method Validator

To create an attribute-level Method validator:

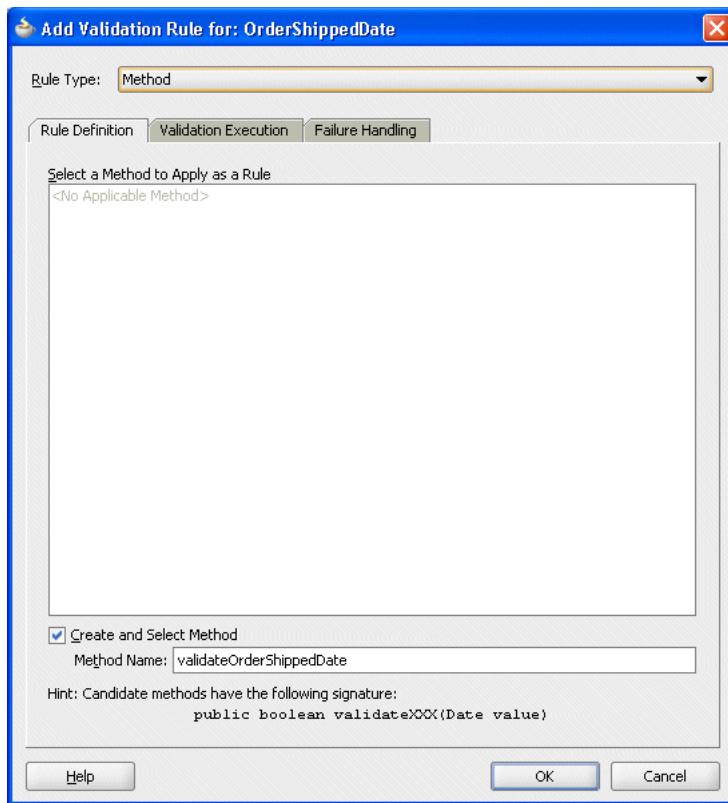
1. In the Application Navigator, double-click the desired entity object.
2. In the overview editor, click the **Java** tab.

The Java page shows the Java generation options that are currently enabled for the entity object. If your entity object does not yet have a custom entity object class, then you must generate one before you can add a Method validator. To generate the custom Java class, click the **Edit** icon, then select **Generate Entity Object Class**, and click **OK** to generate the *.java file.

3. Click the **Validators** tab, and then expand the **Attributes** node, and select the attribute that you want to validate.
4. Click the **New** icon to add a validation rule.
5. Select **Method** from the **Rule Type** dropdown list.

The Add Validation Rule dialog displays the expected method signature for an attribute-level validation method. You have two choices:

- If you already have a method in your entity object's custom Java class of the appropriate signature, it will appear in the list and you can select it after deselecting the **Create and Select Method** checkbox.
 - If you leave the **Create and Select Method** checkbox selected (see [Figure 8–2](#)), you can enter any method name in the **Method Name** box that begins with the word validate. When you click **OK**, JDeveloper adds the method to your entity object's custom Java class with the appropriate signature.
6. Finally, supply the text of the error message for the default locale that the end user should see if this validation rule fails.

Figure 8–2 Adding an Attribute-Level Method Validator

8.2.2 What Happens When You Create an Attribute-Level Method Validator

When you add a new method validator, JDeveloper updates the XML component definition to reflect the new validation rule. If you asked to have the method created, the method is added to the entity object's custom Java class. [Example 8–2](#) illustrates a simple attribute-level validation rule that ensures that the `OrderShippedDate` of an order is a date in the current month. Notice that the method accepts an argument of the same type as the corresponding attribute, and that its conditional logic is based on the value of this incoming parameter. When the attribute validator fires, the attribute value has not yet been set to the new value in question, so calling the `getOrderShippedDate()` method inside the attribute validator for the `OrderShippedDate` attribute would return the attribute's *current* value, rather than the *candidate* value that the client is attempting to set.

Example 8–2 Simple Attribute-Level Method Validator

```
public boolean validateOrderShippedDate(Date data) {
    if (data != null && data.compareTo(getFirstDayOfCurrentMonth()) <= 0) {
        return false;
    }
    return true;
}
```

Note: The return value of the `compareTo()` method is zero (0) if the two dates are equal, negative one (-1) if the first date is less than the second, or positive one (1) if the first date is greater than the second.

8.2.3 How to Create an Entity-Level Method Validator

To create an entity-level method validator:

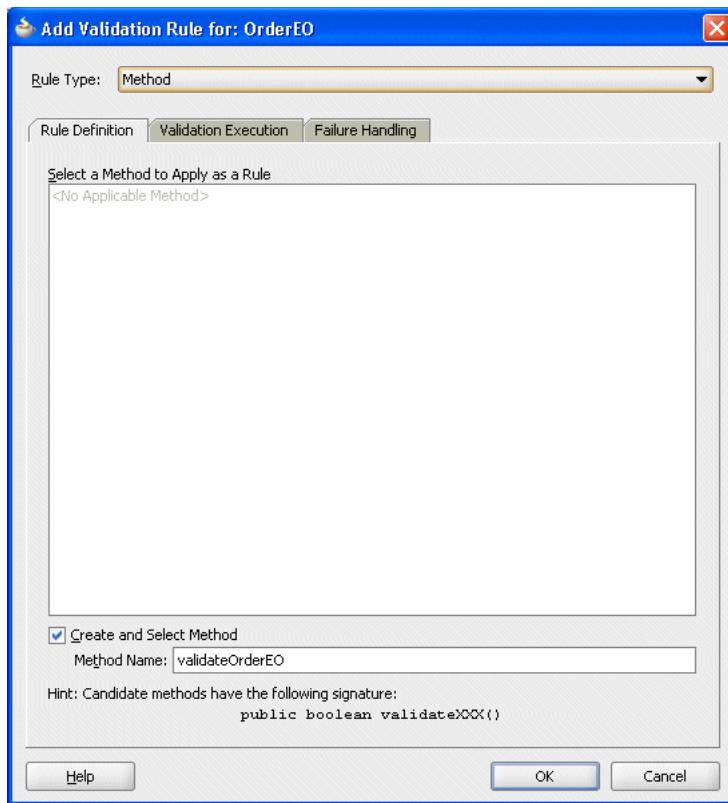
1. In the Application Navigator, double-click the desired entity object.
2. In the overview editor, click the **Java** tab.

The Java page shows the Java generation options that are currently enabled for the entity object. If your entity object does not yet have a custom entity object class, then you must generate one before you can add a Method validator. To generate the custom Java class, click the **Edit** icon, then select **Generate Entity Object Class**, and click **OK** to generate the *.java file.

3. Click the **Validators** tab, and then select the **Entity** node.
4. Click the **New** icon to add a validation rule.
5. Select **Method** from the **Rule Type** dropdown list.

The Add Validation Rule dialog displays the expected method signature for an entity-level validation method. You have two choices:

- If you already have a method in your entity object's custom Java class of the appropriate signature, it will appear in the list and you can select it after deselecting the **Create and Select Method** checkbox.
 - If you leave the **Create and Select Method** checkbox selected (see [Figure 8-3](#)), you can enter any method name in the **Method Name** box that begins with the word validate. When you click **OK**, JDeveloper adds the method to your entity object's custom Java class with the appropriate signature.
6. Finally, supply the text of the error message for the default locale that the end user should see if this validation rule fails.

Figure 8–3 Adding an Entity-Level Method Validator

8.2.4 What Happens When You Create an Entity-Level Method Validator

When you add a new method validator, JDeveloper updates the XML component definition to reflect the new validation rule. If you asked to have the method created, the method is added to the entity object's custom Java class. [Example 8–3](#) illustrates a simple entity-level validation rule that ensures that the OrderShippedDate of an order comes after the OrderDate.

Example 8–3 Simple Entity-Level Method Validator

```
public boolean validateOrderShippedDateAfterOrderDate() {
    Date orderShippedDate = getOrderShippedDate();
    Date orderDate = getOrderDate();
    if (orderShippedDate != null && orderShippedDate.compareTo(orderDate) < 0) {
        return false;
    }
    return true;
}
```

8.2.5 What You May Need to Know About Translating Validation Rule Error Messages

Like the locale-specific UI control hints for entity object attributes, the validation rule error messages are added to the entity object's component message bundle file. These entries in the message bundle represent the strings for the default locale for your application. To provide translated versions of the validation error messages, follow the same steps as for translating the UI control hints, as described in [Section 4.7, "Working with Resource Bundles"](#).

8.3 Assigning Programmatically Derived Attribute Values

When declarative defaulting falls short of your needs, you can perform programmatic defaulting in your entity object:

- When an entity row is first created
- When the entity row is first created or when refreshed to null values
- When the entity row is saved to the database
- When an entity attribute value is set

8.3.1 How to Provide Default Values for New Rows at Create Time

The `create()` method provides the entity object event you can handle to initialize default values the first time an entity row is created. [Example 8–4](#) shows the overridden `create` method of the `OrderEO` entity object in the `StoreFront` module of the Fusion Order Demo. It calls an attribute setter methods to populate the `OrderDate` attribute in a new order entity row.

You can also define default values using a Groovy expression. For more information, see [Section 4.10.6, "How to Define a Static Default Value"](#).

Example 8–4 Programmatically Defaulting Attribute Values for New Rows

```
// In OrderEOImpl.java in Fusion Order Demo
protected void create(AttributeList nameValuePair) {
    super.create(nameValuePair);
    this.setOrderDate(new Date());
}
```

Note: Calling the `setAttribute()` method inside the overridden `create()` method does not mark the new row as being changed by the user. These programmatically assigned defaults behave like declaratively assigned defaults.

8.3.1.1 Choosing Between `create()` and `initDefaultExpressionAttributes()` Methods

You should override the `initDefaultExpressionAttributes()` method for programmatic defaulting logic that you want to fire both when the row is first created, and when it might be refreshed back to initialized status.

If an entity row has `New` status and you call the `refresh()` method on it, then the entity row is returned to an `Initialized` status if you do not supply either the `REFRESH_REMOVE_NEW_ROWS` or `REFRESH_FORGET_NEW_ROWS` flag. As part of this process, the entity object's `initDefaultExpressionAttributes()` method is invoked, but not its `create()` method again.

8.3.1.2 Eagerly Defaulting an Attribute Value from a Database Sequence

[Section 4.10.9, "How to Synchronize with Trigger-Assigned Values"](#) explains how to use the `DBSequence` type for primary key attributes whose values need to be populated by a database sequence at *commit* time. Sometimes you may want to eagerly allocate a sequence number at entity row creation time so that the user can see its value and so that this value does not change when the data is saved. To accomplish this, use the `SequenceImpl` helper class in the `oracle.jbo.server` package in an overridden `create()` method as shown in [Example 8–5](#). It shows code from the

custom Java class of the `WarehouseEO` entity object in the `StoreFront` module of the `Fusion Order Demo`. After calling `super.create()`, it creates a new instance of the `SequenceImpl` object, passing the sequence name and the current transaction object. Then it calls the `setWarehouseId()` attribute setter method with the return value from `SequenceImpl`'s `getSequenceNumber()` method.

Note: For a metadata-driven alternative to this approach, see [Section 4.11.5, "Assigning the Primary Key Value Using an Oracle Sequence"](#).

Example 8–5 Eagerly Defaulting an Attribute's Value from a Sequence at Create Time

```
// In WarehouseEOImpl.java
import oracle.jbo.server.SequenceImpl;
// Default WarehouseId value from WAREHOUSE_SEQ sequence at entity row create time
protected void create(AttributeList attributeList) {
    super.create(attributeList);
    SequenceImpl sequence = new SequenceImpl("WAREHOUSE_SEQ", getDBTransaction());
    setWarehouseId(sequence.getSequenceNumber());
}
```

8.3.2 How to Assign Derived Values Before Saving

If you want to assign programmatic defaults for entity object attribute values before a row is saved, override the `prepareForDML()` method and call the appropriate attribute setter methods to populate the derived attribute values. To perform the assignment only during `INSERT`, `UPDATE`, or `DELETE`, you can compare the value of the `operation` parameter passed to this method against the integer constants `DML_INSERT`, `DML_UPDATE`, `DML_DELETE` respectively.

[Example 8–6](#) shows an overridden `prepareForDML()` method that assigns derived values.

Example 8–6 Assigning Derived Values Before Saving Using PrepareForDML

```
protected void prepareForDML(int operation, TransactionEvent e) {
    super.prepareForDML(operation, e);
    //Populate GL Date
    if (operation == DML_INSERT) {
        if (this.getGlDate() == null) {
            String glDateDefaultOption =
                (String)this.getInvoiceOption().getAttribute("DefaultGlDateBasis");
            if ("I".equals(glDateDefaultOption)) {
                setAttribute(GLDATE, this.getInvoiceDate());
            } else {
                setAttribute(GLDATE, this.getCurrentDBDate());
            }
        }
    }

    //Populate Exchange Rate and Base Amount if null
    if ((operation == DML_INSERT) || (operation == DML_UPDATE)) {
        BigDecimal defaultExchangeRate = new BigDecimal(1.5);
        if ("Y".equals(this.getInvoiceOption().getAttribute("UseForeignCurTrx"))) {
            if (!(this.getInvoiceCurrencyCode().equals(
                this.getLedger().getAttribute("CurrencyCode")))) {
                if (this.getExchangeDate() == null) {
```

```
        setAttribute(EXCHANGEDATE, this.getInvoiceDate());
    }
    if (this.getExchangeRateType() == null) {
        String defaultConvRateType =
            (String)this.getInvoiceOption().getAttribute("DefaultConvRateType");
        if (defaultConvRateType != null) {
            setAttribute(EXCHANGERATETYPE, defaultConvRateType);
        } else {
            setAttribute(EXCHANGERATETYPE, "User");
        }
    }
    if (this.getExchangeRate() == null) {
        setAttribute(EXCHANGERATE, defaultExchangeRate);
    }
    if ((this.getExchangeRate() != null) &&
        (this.getInvoiceAmount() != null)) {
        setAttribute(INVAMOUNTFUNCCURR,
                    (this.getExchangeRate().multiply(this.getInvoiceAmount())));
    }
} else {
    setAttribute(EXCHANGEDATE, null);
    setAttribute(EXCHANGERATETYPE, null);
    setAttribute(EXCHANGERATE, null);
    setAttribute(INVAMOUNTFUNCCURR, null);
}
}
```

8.3.3 How to Assign Derived Values When an Attribute Value is Set

To assign derived attribute values whenever another attribute's value is set, add code to the latter attribute's setter method. [Example 8-7](#) shows the setter method for an `AssignedTo` attribute in an entity object.

Example 8-7 Setting the Assigned Date Whenever the AssignedTo Attribute Changes

```
public void setAssignedTo(Number value) {  
    setAttributeInternal(ASSIGNEDTO, value);  
    setAssignedDate(getCurrentDateWithTime());  
}
```

After the call to `setAttributeInternal()` to set the value of the `AssignedTo` attribute, it uses the setter method for the `AssignedDate` attribute to set its value to the current date and time.

Note: It is safe to add custom code to the generated attribute getter and setter methods as shown here. When JDeveloper modifies code in your class, it intelligently leaves your custom code in place.

8.4 Undoing Pending Changes to an Entity Using the Refresh Method

You can use the `refresh(int flag)` method on a row to refresh any pending changes it might have. The behavior of the `refresh()` method depends on the flag that you pass as a parameter. The three key flag values that control its behavior are the following constants in the `Row` interface:

- REFRESH_WITH_DB_FORGET_CHANGES forgets modifications made to the row in the current transaction, and the row's data is refreshed from the database. The latest data from the database replaces data in the row regardless of whether the row was modified or not.
- REFRESH_WITH_DB_ONLY_IF_UNCHANGED works just like REFRESH_WITH_DB_FORGET_CHANGES, but for *unmodified* rows. If a row was already modified by this transaction, the row is not refreshed.
- REFRESH_UNDO_CHANGES works the same as REFRESH_WITH_DB_FORGET_CHANGES for *unmodified* rows. For a modified row, this mode refreshes the row with attribute values at the beginning of this transaction. The row remains in a modified state.

8.4.1 How to Control What Happens to New Rows During a Refresh

By default, any entity rows with New status that you `refresh()` are reverted back to blank rows in the Initialized state. Declarative defaults are reset, as well as programmatic defaults coded in the `initDefaultExpressionAttributes()` method, but the entity object's `create()` method is not invoked during this blanking-out process.

You can change this default behavior by combining one of the flags in [Section 8.4](#) with one of the following two flags (using the bitwise-OR operator):

- REFRESH_REMOVE_NEW_ROWS, new rows are removed during refresh.
- REFRESH_FORGET_NEW_ROWS, new rows are marked Dead.

8.4.2 How to Cascade Refresh to Composed Children Entity Rows

You can cause a `refresh()` operation to cascade to composed child entity rows by combining the REFRESH_CONTAINERS flag (using the bitwise-OR operator) with any of the valid flag combinations described in [Section 8.4](#) and [Section 8.4.1](#). This causes the entity to invoke `refresh()` using the same mode on any composed child entities it contains.

8.5 Using View Objects for Validation

When your business logic requires performing SQL queries, the natural choice is to use a view object to perform that task. Keep in mind that the SQL statements you execute for validation will "see" pending changes in the entity cache only if they are entity-based view objects. Read-only view objects will only retrieve data that has been posted to the database.

8.5.1 How to Use View Accessors for Validation Against View Objects

Since entity objects are designed to be reused in any application scenario, they should not depend *directly* on a view object instance in any specific application module's data model. Doing so would prevent them from being reused in other application modules, which is highly undesirable.

Instead, you should use a view accessor to validate against a view object. For more information, see [Section 10.4.1, "How to Create a View Accessor for an Entity Object or View Object"](#).

8.5.2 How to Validate Conditions Related to All Entities of a Given Type

The `beforeCommit()` method is invoked on *each* entity row in the pending changes list after the changes have been posted to the database, but before they are committed. This can be a useful method in which to execute view object based validations that must assert some rule over all entity rows of a given type.

Note: You can also do this declaratively using a transaction-level validator (see [Section 7.6.3, "How to Set Transaction-Level Validation"](#)).

If your `beforeCommit()` logic can throw a `ValidationException`, you must set the `jbo.txn.handleafterpostexc` property to `true` in your configuration to have the framework automatically handle rolling back the in-memory state of the other entity objects that may have already successfully posted to the database (but not yet been committed) during the current commit cycle.

8.6 Accessing Related Entity Rows Using Association Accessors

To access information from related entity objects, you use an association accessor method in your entity object's custom Java class. By calling the accessor method, you can easily access any related entity row — or set of entity rows — depending on the cardinality of the association.

8.6.1 How to Access Related Entity Rows

[Example 8–8](#) shows code from the `ControllingPostingOrder` project in the `AdvancedEntityExamples` module of the Fusion Order Demo that shows the overridden `postChanges()` method in the `ProductsBase` entity object's custom Java class. It uses the `getSupplier()` association accessor to retrieve the related supplier for the product.

Example 8–8 Accessing a Parent Entity Row In a Create Method

```
// In ProductsBaseImpl.java in the ControllingPostingOrder project
// of the Fusion Order Demo Advanced Entity Examples
@Override
public void postChanges(TransactionEvent transactionEvent) {
    /* If current entity is new or modified */
    if (getPostState() == STATUS_NEW || getPostState() == STATUS_MODIFIED) {
        /* Get the associated supplier for the product */
        SuppliersImpl supplier = getSupplier();
        /* If there is an associated product */
        if (supplier != null) {
            /* And if it's post-status is NEW */
            if (supplier.getPostState() == STATUS_NEW) {
                /* Post the supplier first, before posting this entity */
                supplier.postChanges(transactionEvent);
            }
        }
    }
    super.postChanges(transactionEvent);
}
```

8.6.2 How to Access Related Entity RowSets of Rows

If the cardinality of the association is such that multiple rows are returned, you can use the association accessor to return sets of entity rows.

[Example 8–9](#) illustrates the code for the overridden `postChanges()` method in the `Suppliers` entity object's custom Java class. It shows the use of the `getProductsBase()` association accessor to retrieve the RowSet of `ProductsBase` rows in order to update the `SupplierId` attribute in each row using the `setSupplierId()` association accessor.

Example 8–9 Accessing a Related Entity RowSet Using an Association Accessor

```
// In SuppliersImpl.java in the ControllingPostingOrder project
// of the Fusion Order Demo Advanced Entity Examples
RowSet newProductsBeforePost = null;
@Override
public void postChanges(TransactionEvent transactionEvent) {
    /* Only update references if Supplier is new */
    if (getPostState() == STATUS_NEW) {
        /*
         * Get a rowset of products related to this new supplier before calling super
         */
        newProductsBeforePost = (RowSet) getProductsBase();
    }
    super.postChanges(transactionEvent);
}

...
protected void refreshFKInNewContainees() {
    if (newProductsBeforePost != null) {
        Number newSupplierId = getSupplierId().getSequenceNumber();
        /*
         * Process the rowset of suppliers that referenced the new product prior
         * to posting, and update their ProdId attribute to reflect the refreshed
         * ProdId value that was assigned by a database sequence during posting.
         */
        while (newProductsBeforePost.hasNext()) {
            ProductsBaseImpl product = (ProductsBaseImpl) newProductsBeforePost.next();
            product.setSupplierId(newSupplierId);
        }
        closeNewProductRowSet();
    }
}
```

8.7 Referencing Information About the Authenticated User

If you have set the `jbo.security.enforce` runtime configuration property to the value `Must` or `Auth`, the `oracle.jbo.server.SessionImpl` object provides methods you can use to get information about the name of the authenticated user and about the roles of which they are a member. This is the implementation class for the `oracle.jbo.Session` interface that clients can access.

The following topics explain how to access information about the authenticated user:

- [Section 28.7.3, "How to Determine the Current User Name"](#)
- [Section 28.7.4, "How to Determine Membership of a Java EE Security Role"](#)

For more information about security features in Oracle Fusion Web Applications, read [Chapter 28, "Adding Security to a Fusion Web Application"](#).

8.8 Accessing Original Attribute Values

If an entity attribute's value has been changed in the current transaction, when you call the attribute getter method for it you will get the pending changed value. Sometimes you want to get the original value before it was changed. Using the `getPostedAttribute()` method, your entity object business logic can consult the original value for any attribute as it was read from the database before the entity row was modified. This method takes the attribute *index* as an argument, so pass the appropriate generated attribute index constants that JDeveloper maintains for you.

8.9 Storing Information About the Current User Session

If you need to store information related to the current user session in a way that entity object business logic can reference, you can use the user data hash table provided by the `Session` object.

8.9.1 How to Store Information About the Current User Session

In the following examples, when a new user accesses an application module for the first time, the `prepareSession()` method is called. As shown in [Example 8–10](#), the application module overrides `prepareSession()` to retrieve information about the authenticated user by calling a `retrieveUserInfoForAuthenticatedUser()` method on the view object instance. Then, it calls the `setUserIdIntoUserDataHashtable()` helper method to save the user's numerical ID into the user data hash table.

Example 8–10 Overriding `prepareSession()` to Query User Information

```
// In the application module
protected void prepareSession(Session session) {
    super.prepareSession(session);
    /*
     * Query the correct row in the VO based on the currently logged-in
     * user, using a custom method on the view object component
     */
    getLoggedInUser().retrieveUserInfoForAuthenticatedUser();
    setUserIdIntoUserDataHashtable();
}
```

[Example 8–11](#) shows the code for the view object's `retrieveUserInfoForAuthenticatedUser()` method. It sets its own `EmailAddress` bind variable to the name of the authenticated user from the session and then calls `executeQuery()` to retrieve the additional user information from the `USERS` table.

Example 8–11 Accessing Authenticated User Name to Retrieve Additional User Details

```
// In the view object's custom Java class
public void retrieveUserInfoForAuthenticatedUser() {
    SessionImpl session = (SessionImpl) getDBTransaction().getSession();
    setEmailAddress(session.getUserPrincipalName());
    executeQuery();
    first();
}
```

One of the pieces of information about the authenticated user that the view object retrieves is the user's numerical ID number, which that method returns as its result. For example, the user sking has the numeric UserId of 300.

[Example 8–12](#) shows the `setUserIdIntoUserDataHashtable()` helper method — used by the `prepareSession()` code in [Example 8–10](#) — that stores this numerical user ID in the user data hash table, using the key provided by the string constant `CURRENT_USER_ID`.

Example 8–12 Setting Information into the UserData Hashtable for Access By Entity Objects

```
// In the application module
private void setUserIdIntoUserDataHashtable() {
    Integer userid = getUserIdForLoggedInUser();
    Hashtable userdata = getDBTransaction().getSession().getUserData();
    userdata.put(CURRENT_USER_ID, userid);
}
```

The corresponding entity objects in this example can have an overridden `create()` method that references this numerical user ID using a helper method like the one in [Example 8–13](#) to set the `CreatedBy` attribute programmatically to the value of the currently authenticated user's numerical user ID.

Example 8–13 Referencing the Current User ID in a Helper Method

```
protected Number getCurrentUserId() {
    Hashtable userdata = getDBTransaction().getSession().getUserData();
    Integer userId = (Integer)userdata.get(CURRENT_USER_ID);
    return userdata != null ? Utils.intToNumber(userId):null;
}
```

8.10 Accessing the Current Date and Time

You might find it useful to reference the current date and time in your entity object business logic. You can reference the current date or current date and time using the following Groovy script expressions:

- `adf.currentDate` — returns the current date (time truncated)
- `adf.currentTime` — returns the current date and time

For more information about using Groovy script in your entity object business logic, see [Section 3.6, "Overview of Groovy Support"](#).

8.11 Sending Notifications Upon a Successful Commit

The `afterCommit()` method is invoked on each entity row that was in the pending changes list and got successfully saved to the database. You can use this method to send a notification on a commit.

8.12 Conditionally Preventing an Entity Row from Being Removed

Before an entity row is removed, the `remove()` method is invoked on an entity row.

You can throw a `JboException` in the `remove()` method to prevent a row from being removed if the appropriate conditions are not met.

Note: The entity object offers declarative prevention of deleting a master entity row that has existing, composed children rows. You configure this option on the Relationship page of the overview editor for the association.

8.13 Determining Conditional Updatability for Attributes

You can override the `isAttributeUpdateable()` method in your entity object class to programmatically determine whether a given attribute is updatable or not at runtime based on appropriate conditions.

[Example 8-14](#) shows how an entity object can override the `isAttributeUpdateable()` method to enforce that its `PersonTypeCode` attribute is updatable only if the current authenticated user is a staff member. Notice that when the entity object fires this method, it passes in the integer attribute index whose updatability is being considered.

You can implement conditional updatability logic for a particular attribute inside an `if` or `switch` statement based on the attribute index. Here `PERSONTYPECODE` is referencing the integer attribute index constants that JDeveloper maintains in your entity object custom Java class.

Example 8-14 Conditionally Determining an Attribute's Updatability at Runtime

```
// In the entity object custom Java class
public boolean isAttributeUpdateable(int index) {
    if (index == PERSONTYPECODE) {
        if (!currentUserIsStaffMember()) {
            return super.isAttributeUpdateable(index);
        }
        return CUSTOMER_TYPE.equals(getPersonTypeCode()) ? false : true;
    }
    return super.isAttributeUpdateable(index);
}
```

Note: Entity-based view objects inherit this conditional updatability as they do everything else encapsulated in your entity objects. Should you need to implement this type of conditional updatability logic in a way that is specific to a transient view object attribute, or to enforce some condition that involves data from multiple entity objects participating in the view object, you can override this same method in a view object's view row class to achieve the desired result.

Implementing Business Services with Application Modules

This chapter describes how to create application modules to encapsulate a data model using view objects. This chapter also describes how to combine business service methods with that data model to implement a complete business service.

This chapter includes the following sections:

- [Section 9.1, "Introduction to Application Modules"](#)
- [Section 9.2, "Creating and Modifying an Application Module"](#)
- [Section 9.3, "Configuring Your Application Module Database Connection"](#)
- [Section 9.4, "Defining Nested Application Modules"](#)
- [Section 9.5, "Creating an Application Module Diagram for Your Business Service"](#)
- [Section 9.6, "Supporting Multipage Units of Work"](#)
- [Section 9.7, "Customizing an Application Module with Service Methods"](#)
- [Section 9.8, "Publishing Custom Service Methods to UI Clients"](#)
- [Section 9.9, "Debugging the Application Module Using the Business Component Browser"](#)
- [Section 9.10, "Working Programmatically with an Application Module's Client Interface"](#)
- [Section 9.11, "Overriding Built-in Framework Methods"](#)

9.1 Introduction to Application Modules

An *application module* is an Oracle ADF Business Component component that encapsulates the business service methods and UI-aware data model for a logical unit of work related to an end-user task.

In the early phases of application development, architects and designers often use UML use case techniques to create a high-level description of the application's planned end-user functionalities. Each high-level, end-user use case identified during the design phase typically depends on:

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

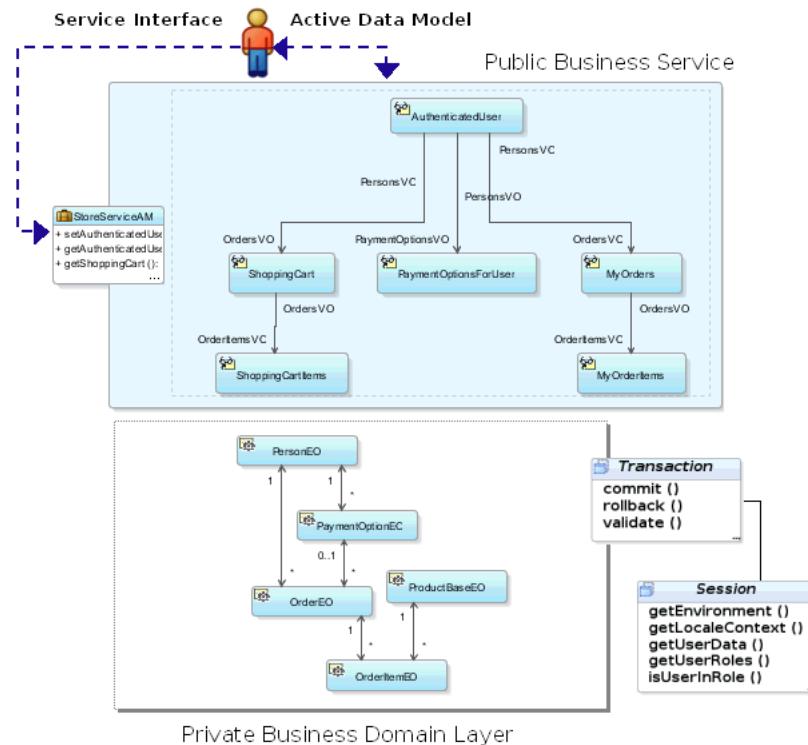
The identified domain objects involved in each use case help you identify the required entity objects from your business domain layer. The user-oriented view of the required business data helps to define the right SQL queries captured as view objects and to retrieve the data in the exact way needed by the end user. For best performance, this includes retrieving the minimum required details necessary to support the use case. In addition to leveraging view object queries to shape the data, you've learned how to use view links to set up natural master-detail hierarchies in your data model to match exactly the kind of end-user experience you want to offer the user to accomplish the use case.

The application module is the "work unit" container that includes instances of the reusable view objects required for the use case in question, related through metadata to the underlying entity objects in your reusable business domain layer whose information the use case is presenting or modifying.

This chapter illustrates the following concepts illustrated in [Figure 9–1](#), and more:

- You use instances of view objects in an application module to define its data model.
- You write service methods to encapsulate task-level business logic.
- You expose selected methods on the client interface for UI clients to call.
- You expose selected methods on the service interface for programmatic use in application integration scenarios.
- You use application modules from a pool during a logical transaction that can span multiple web pages.
- Your application module works with a `Transaction` object that acquires a database connection and coordinates saving or rolling back changes made to entity objects.
- The related `Session` object provides runtime information about the current application user.

Figure 9–1 Application Module Is a Business Service Component Encapsulating a Unit of Work



9.2 Creating and Modifying an Application Module

In a large application, you typically create one application module to support each coarse-grained end-user task. In a smaller-sized application, you may decide that creating a single application module is adequate to handle the needs of the complete set of application functionality. [Section 9.4, "Defining Nested Application Modules"](#) provides additional guidance on this subject.

9.2.1 How to Create an Application Module

Any view object you create is a reusable component that can be used in the context of one or more application modules to perform the query it encapsulates in the context of that application module's transaction. The set of view objects used by an application module defines its *data model*, in other words, the set of data that a client can display and manipulate through a user interface.

To create an application module, use the Create Application Module wizard, which is available in the New Gallery.

To create an application module:

1. In the Application Navigator, right-click the project in which you want to create the application module and choose **New**.
2. In the New Gallery, expand **Business Tier**, select **ADF Business Components**, and then **Application Module**, and click **OK**.
3. In the Items list, select **Application Module**.
4. In the Create Application Module wizard, on the Name page, provide a package name and an application module name. Click **Next**.

Note: In Fusion web applications, the reserved words `data`, `bindings`, `security`, and `adfContext` must not be used to name your application module. Also, avoid using the `"_"` (underscore) at the beginning of the name. For more information, see [Section 9.2.5, "How to Edit an Existing Application Module"](#).

5. On the Data Model page, include instances of the view objects you have previously defined and edit the view object instance names to be exactly what you want clients to see. Then click **Next**.
6. On the Java page, you can optionally generate the Java files that allow you to programmatically customize the behavior of the application module or to expose methods on the application module's client interface that can be called by clients. To generate an XML-only application module component, leave the fields unselected and click **Finish**.

Initially, you may want to generate only the application module XML definition component. After you complete the wizard, you can subsequently use the overview editor to generate the application module class files when you require programmatic access. For details about the programmatic use of the application module, see [Section 9.7, "Customizing an Application Module with Service Methods"](#).

For more step by step details, see [Section 9.2.3, "How to Add a View Object to an Application Module"](#).

9.2.2 What Happens When You Create an Application Module

When you create an application module, JDeveloper creates the XML component definition file that represents its declarative settings and saves it in the directory that corresponds to the name of its package. For example, given an application module named `StoreServiceAM` in the `storefront.model` package, the XML file created will be `./storefront/model/StoreServiceAM.xml` under the project's source path. This XML file contains the information needed at runtime to recreate the view object instances in the application module's data model.

If you are curious to view its contents, you can see the XML file for the application module by double-clicking the `StoreServiceAM` node in the Application Navigator to open the editor window. In the editor, click the **Source** tab to view the XML in an editor so that you can inspect it. The Structure window shows the structure of the XML file.

Note: The wizard will also let you create a custom application module class. For example, the file `StoreServiceAMImpl.java`. This file is optional.

9.2.3 How to Add a View Object to an Application Module

You can add a view object to an application module as you are creating the application module with the Create Application Module wizard, or you can add it later. To add a view object to an application module, and optionally, customize the view object instance, use the Data Model page of the Edit Application Module dialog.

For information about using the Create Application Module wizard, see [Section 9.2.1, "How to Create an Application Module"](#).

To add a view object instance to an existing application module:

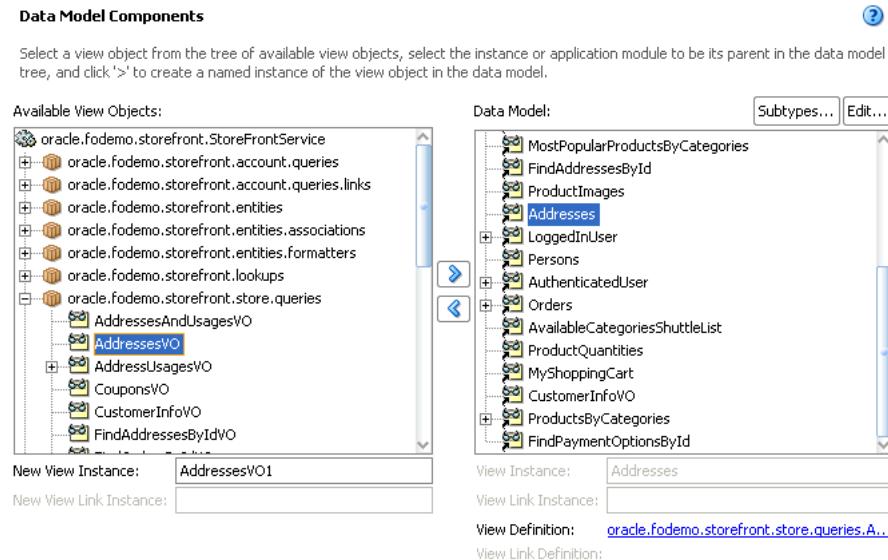
1. In the Application Navigator, double-click the application module.
2. In the overview editor, select the **Data Model** navigation tab.
3. On the Data Model Components page, in the **Available View Objects** list, select the view instance you want to add.

The **New View Instance** field below the list shows the name that will be used to identify the next instance of that view object that you add to the data model.

4. To change the name before adding it, enter a different name in the **New View Instance** field.
5. With the desired view object selected, shuttle the view object to the **Data Model** list.

[Figure 9–2](#) shows the view object `AddressVO` has been renamed to `Address` before it was shuttled to the **Data Model** list.

Figure 9–2 Data Model Displays Added View Object Instances



You can use the data model that the application module overview editor displays to create a hierarchy of view instances, based on existing view links that your project defines. If you have defined view links that establish more than one level of master-detail hierarchy, then you can proceed to create as many levels of master-detail view instances as your application supports. For details about creating hierarchical relationships using view links, see [Section 5.6, "Working with Multiple Tables in a Master-Detail Hierarchy"](#).

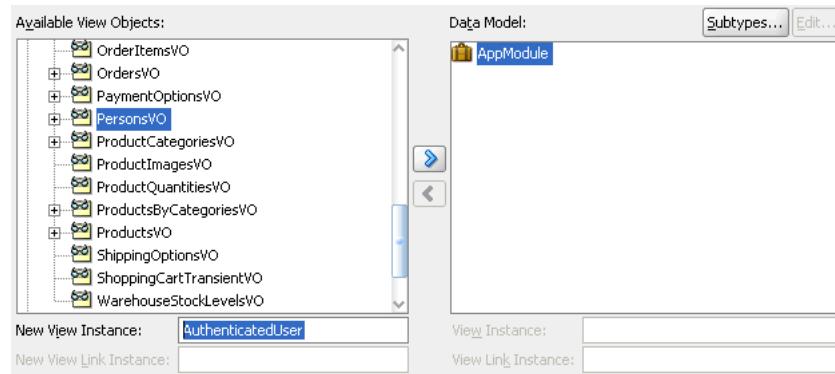
To add master-detail view object instances to a data model:

1. In the Application Navigator, double-click the application module.
2. In the overview editor, select the **Data Model** navigation tab.
3. On the Data Model Components page, in the **Data Model** list on the right, select the instance of the view object that you want to be the actively coordinating master.

The master view object will appear with a plus sign in the list indicating the available view links for this view object. The view link must exist to define a master-detail hierarchy.

[Figure 9–3](#) shows PersonsVO selected and renamed AuthenticatedUser in the **New View Instance** field.

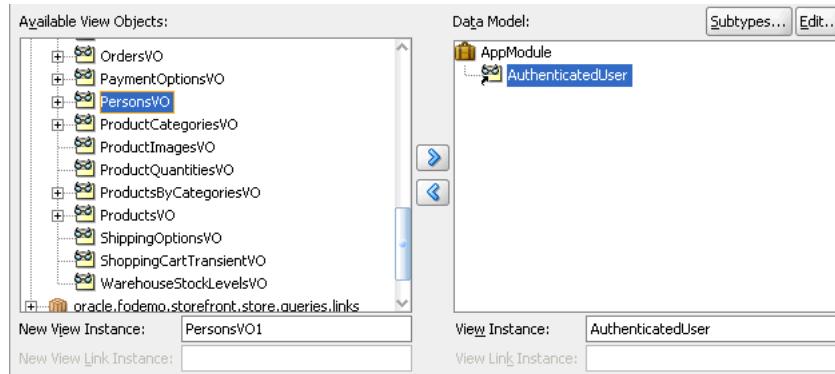
Figure 9–3 Master View Object Selected



- Shuttle the selected master view object to the **Data Model** list

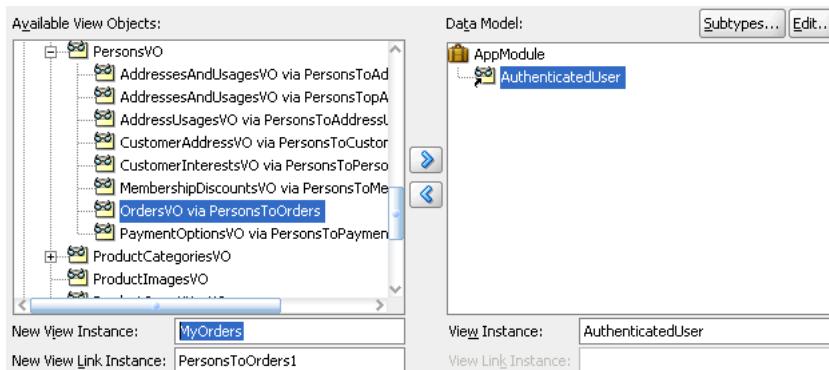
[Figure 9–4](#) shows the newly created master view instance AuthenticatedUser in the **Data Model** list.

Figure 9–4 Master View Instance Created



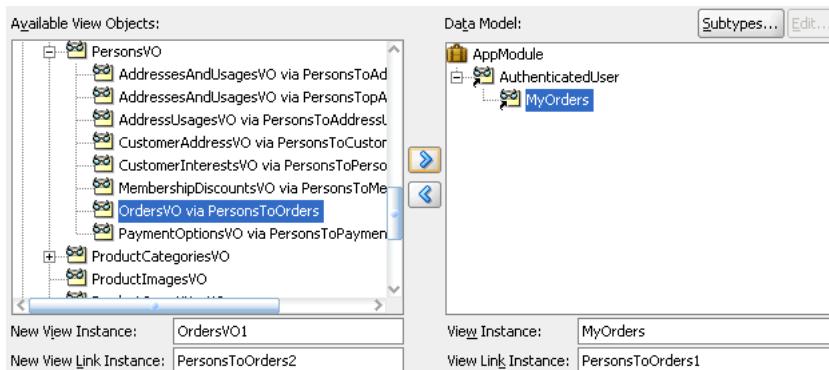
- In the **Data Model** list, leave the newly created master view instance selected so that it appears highlighted. This will be the target of the detail view instance you will add. Then locate and select the detail view object beneath the master view object in the **Available View Objects** list.

[Figure 9–5](#) shows the detail OrdersVO indented beneath master PersonsVO with the name OrdersVO via PersonsToOrders. The name identifies the view link PersonsToOrders, which defines the master-detail hierarchy between PersonsVO and OrdersVO. Notice also that the OrdersVO will have the view instance name MyOrders when added to the data model.

Figure 9–5 Detail View Object Selected

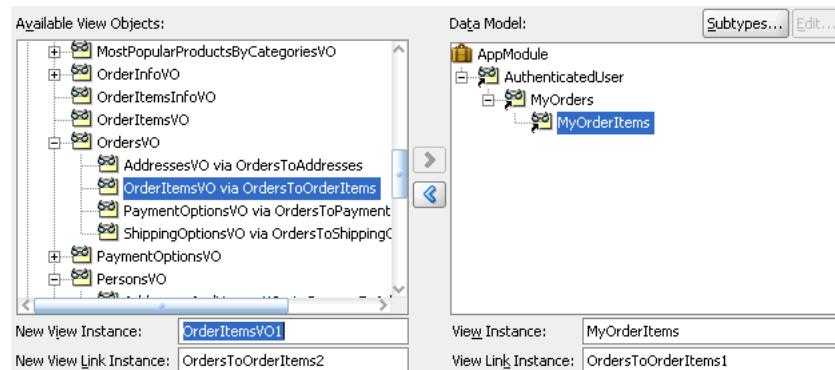
6. To add the detail instance to the previously added master instance, shuttle the detail view object to the **Data Model** list below the selected master view instance.

Figure 9–6 shows the newly created detail view instance `MyOrders` as a detail of the `AuthenticatedUser` in the data model.

Figure 9–6 Master View Instance Created

7. To add another level of hierarchy, repeat Step 3 through Step 6, but select the newly added detail in the **Data Model** list, then shuttle over the new detail, which itself has a master-detail relationship with the previously added detail instance.

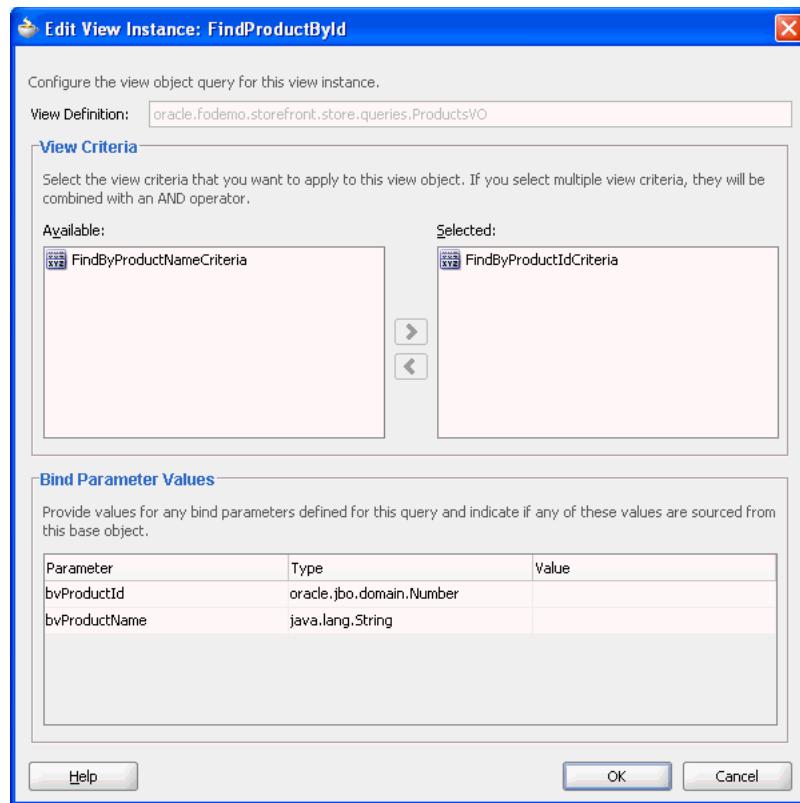
Figure 9–7 shows the **Data Model** list with instance `AuthenticatedUser` (renamed from `PersonsVO`) as the master of `MyOrders` (renamed from `OrdersVO via PersonsToOrders`), which is, in turn, a master for `MyOrderItems` (renamed from `OrderItemsVO via OrdersToOrderItems`).

Figure 9–7 Master-Detail-Detail Hierarchy Created

To customize a view object instance that you add to an existing application module:

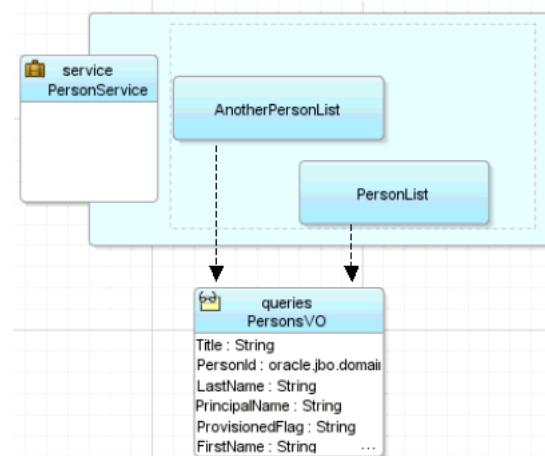
1. In the Application Navigator, double-click the application module.
2. In the overview editor, select the **Data Model** navigation tab.
3. On the Data Model Components page, in the **Data Model** list, select the view object instance you want to customize and click **Edit**.
4. In the Edit View Instance dialog, perform any of the following steps, and then click **OK**.
 - In the **View Criteria** group box, select one or more view criteria that you want to apply to the view object instance. The view criteria will be appended as a WHERE clause to the instance query. For details about defining view criteria, see [Section 5.10, "Working with Named View Criteria"](#).
 - In the **Bind Parameters Values** group box, enter any values that you wish the instance to use when applying the defined view criteria. For more information about defining bind variables, see [Section 5.9, "Working with Bind Variables"](#).

[Figure 9–8](#) shows the Edit View Instance dialog with the `FindByProductIdCriteria` selected. No default value is supplied for the bind variables `bvProductId` and `bvProductName` since the values will be provided as a search criteria by the user at runtime.

Figure 9–8 Customized View Object Instances Using a View Criteria Filter

9.2.4 What Happens When You Add a View Object to an Application Module

While adding a view object to an application module, you use instances of a view object component to define its data model. [Figure 9–9](#) shows a JDeveloper business components diagram of a PersonService application module.

Figure 9–9 Application Module Containing Two Instances of a View Object Component

The sample application module contains two instances of the Persons view object component, with member names of PersonList and AnotherPersonList to distinguish them. At runtime, both instances share the same PersonsVO view object component definition—ensure that they have the same attribute structure and view

object behavior—however, each might be used independently to retrieve data about different users. For example, some of the runtime properties, like an additional filtering WHERE clause or the value of a bind variable, might be different on the two distinct instances.

[Example 9–1](#) shows how the PersonService application module defines its member view object instances in its XML component definition file.

Example 9–1 Member View Object Instances Defined in XML

```
< AppModule Name="PersonService">
    < ViewUsage
        Name="PersonList"
        ViewObjectName="oracle.fodemo.storefront.store.queries.PersonsVO" />
    < ViewUsage
        Name="AnotherPersonList"
        ViewObjectName="oracle.fodemo.storefront.store.queries.PersonsVO" />
</ AppModule>
```

9.2.5 How to Edit an Existing Application Module

After you've created a new application module, you can edit any of its settings by using the Edit Application Module dialog. To launch the editor, choose **Open** from the context menu in the Application Navigator, or double-click the application module. By visiting the different pages of the editor, you can adjust the data model to determine whether or not to reference nested application modules, specify Java generation settings, client interface methods, remote deployment options, runtime instantiation behavior, and custom properties.

If you edit the name of your application module, choose a name that is not among the reserved words that Oracle ADF defines. In particular, reserved words are not valid for a data control usage name which JDeveloper automatically assigns based on your application module's name. In Fusion web applications, these reserved words consist of data, bindings, security, and adfContext. For example, you should not name an application module data. If JDeveloper creates a data control usage with an ID that collides with a reserved word, your application may not reliably access your data control objects at runtime and may fail with a runtime `ClassCastException`.

Do not name the application module with an initial underscore (_) character to prevent a potential name collision with a wider list of reserved words that begin with the underscore.

Application module names that incorporate a reserved word into their name (or that change the case of the reserved word) will not conflict. For example, `Product_Data`, `Product_data`, or just `Data` are all valid application module names since the whole name does not match the reserved word `data`.

9.2.6 How to Change the Data Control Name Before You Begin Building Pages

By default, an application module will appear in the Data Controls panel as a data control named `AppModuleDataControl`. The user interface designer uses the Data Controls panel to bind data from the application module to the application's web pages. For example, if the application module is named `StoreServiceAM`, the Data Controls panel will display the data control with the name `StoreServiceAMDataControl`. You can change the default data control name to make it shorter or to supply a more preferable name.

When the user interface designer works with the data control, they will see the data control name for your application module in the `DataBindings.cpx` file in the user interface project and in each data binding page definition XML file. In addition, you might refer to the data control name in code when needing to work programmatically with the application module service interface. For this reason, if you plan to change the name of your application module, do this change before you begin building your view layer.

For complete information about the application module data control, see [Chapter 11, "Using Oracle ADF Model in a Fusion Web Application"](#).

Note: If you decide to change the application module's data control name after you have already referenced it in one or more pages, you will need to open the page definition files and `DataBindings.cpx` file where it is referenced and update the old name to the new name manually.

To change the application module data control name:

1. Open the application module in the overview editor.
2. Open the Property Inspector and expand the **Other** section.
3. Enter your preferred data control name in the **Data Control Name** field.

9.2.7 What You May Need to Know About Application Module Granularity

A common question related to application modules is, "How big should my application module be?" In other words, "Should I build one big application module to contain the entire data model for my enterprise application, or many smaller application modules?" The answer depends on your situation.

In general, application modules should be as big as necessary to support the specific use case you have in mind for them to accomplish. They can be assembled from finer-grained application module components using a nesting feature, as described in [Section 9.4, "Defining Nested Application Modules"](#). Since a complex business application is not really a single use case, a complex business application implemented using Oracle ADF will typically not be just a single application module.

In actual practice, you may choose any granularity you wish. For example, in a small application with one main use case and a "backend" supporting use case, you could create two application modules. However, for the sake of simplicity you can combine both use cases, rather than create a second application module that contains just a couple of view objects.

9.2.8 What You May Need to Know About View Object Components and View Object Instances

While designing an application module, you use instances of a view object component to define its data model. Just as the user interface may contain two instances of a `Button` component with member names of `myButton` and `anotherButton` to distinguish them, your application module contains two instances of the `Persons` view object component, with member names of `PersonList` and `AnotherPersonList` to distinguish them.

9.3 Configuring Your Application Module Database Connection

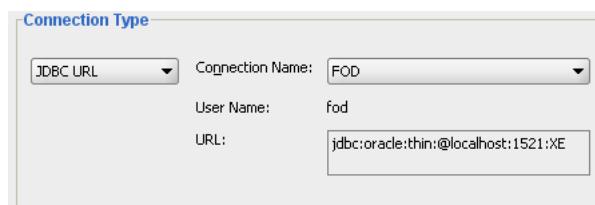
You configure your application module to use a database connection by identifying either a Java Database Connectivity (JDBC) URL or a JDBC data source name in the **Connection Type** section of the Edit Business Components Configuration dialog.

9.3.1 How to Use a JDBC URL Connection Type

The default `YourAppNameLocal` configuration uses a JDBC URL connection. It is based on the named connection definition set on the Business Components page of the Project Properties dialog for the project containing your application module.

[Figure 9–10](#) shows what this section would look like in a configuration using a JDBC URL connection. The advantage of using the default JDBC URL connection type is that it allows you to run the application module in any context where Java can run. In other words, it is not restricted to running inside a Java Enterprise Edition (Java EE) application server with this connection type. The JDBC URL connection type is the one you will use to test your business components in the Business Component Browser since the tester does not support JDBC data sources as a connection type.

Figure 9–10 Connection Type Setting in Edit Business Components Configuration Dialog



Note: See [Section 37.4.2, "Database Connection Pools"](#) and [Section 37.6, "Database Connection Pool Parameters"](#) for more information on how database connection pools are used and how you can tune them.

9.3.2 How to Use a JDBC Data Source Connection Type

The other type of connection you can use is a JDBC data source. You define a JDBC data source as part of your application server configuration information, and then the application module looks up the resource at runtime using a logical name. You define the data source connection details in two parts:

- A logical part that uses standard Java EE deployment descriptors to define the data source name the application will use at runtime
- A physical part that is application-server specific which maps the logical data source name to the physical connection details

[Example 9–2](#) shows the `<resource-ref>` tags in the `web.xml` file of a Fusion web application. These define two logical data sources named `jdbc/FODemoDS` and `jdbc/FODemoCoreDS`. When you use a JDBC data source connection for your application module, you reference this logical connection name after the prefix `java:comp/env`. The configuration for the application module in this same Fusion web application would display the value `java:comp/env/jdbc/FODemoDS` in the **JDBC Datasource Name** field.

Example 9–2 Logical Data Source Resource Names Defined in web.xml

```
<!-- In web.xml -->
<resource-ref>
  <res-ref-name>jdbc/FODemoDS</res-ref-name>
  <res-type>javax.sql.DataSource</res-type>
  <res-auth>Container</res-auth>
</resource-ref>
<resource-ref>
  <res-ref-name>jdbc/FODemoCoreDS</res-ref-name>
  <res-type>javax.sql.DataSource</res-type>
  <res-auth>Container</res-auth>
</resource-ref>
```

Once you've defined the logical data source names in `web.xml` and referenced them in a configuration, you then need to include additional, server-specific configuration files to map the logical data source names to physical connection definitions in the target application server. [Example 9–3](#) shows the contents of the `data-sources.xml` file in a Fusion web application that utilizes a data source connection. The `data-sources.xml` file is specific to the Oracle WebLogic Server and defines the physical details of the data source connection pools and connection names. You would need to provide a similar file for different Java EE application servers, and the file would have a vendor-specific format.

Example 9–3 Data Source Connection Details Defined External to Application

```
<data-sources ... >
  <connection-pool name="jdev-connection-pool-FODemo">
    <connection-factory
      factory-class="oracle.jdbc.pool.OracleDataSource"
      password="->DataBase_User_8PQ34e6j3MDg3UcQD-BZktUAK-QpepGp"
      user="fusionorderdemo" url="jdbc:oracle:thin:@localhost:1521:XE"/>
  </connection-pool>
  <managed-data-source name="jdev-connection-managed-FODemo"
    jndi-name="jdbc/FODemoDS"
    connection-pool-name="jdev-connection-pool-FODemo"/>
  <native-data-source name="jdev-connection-native-FODemo"
    jndi-name="jdbc/FODemoCoreDS" ... />
</data-sources>
```

The last step in the process involves mapping the physical connection details to the logical resource references for the data source. In the Oracle WebLogic Server, you accomplish this step using the `orion-web.xml` file, as shown in [Example 9–4](#).

Example 9–4 Server-Specific File Maps Logical Data Source to Physical Data Source

```
<orion-web-app ... >
  <resource-ref-mapping location="jdbc/FODemoDS" name="jdbc/FODemoDS"/>
  <resource-ref-mapping location="jdbc/FODemoCoreDS" name="jdbc/FODemoCoreDS"/>
</orion-web-app>
```

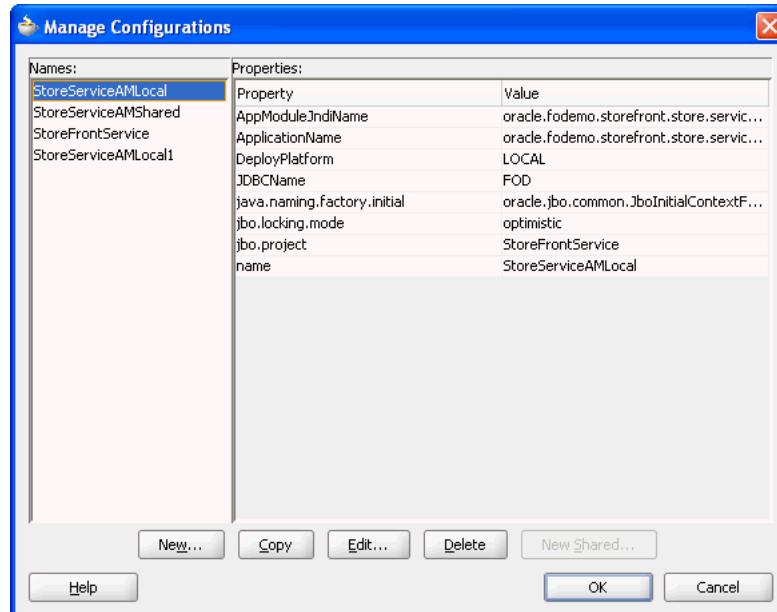
Once these data source configuration details are in place, you can successfully use your application module in a Java EE application server as part of a Fusion web application.

9.3.3 How to Change Your Application Module's Runtime Configuration

In addition to creating the application module XML component definition, JDeveloper also adds a default configuration named `appModuleLocal` to the `bc4j.xcfg`

file in the subdirectory named `common`, relative to the directory containing the application module XML component definition file. The `bc4j.xcfg` file does not appear in the Application Navigator. To view the default settings or to change the application module's runtime configuration settings, you can use the Manage Configurations dialog shown in [Figure 9–11](#).

Figure 9–11 bc4j.xcfg File Configurations Displayed by Manage Configurations Dialog



To manage your application module's configuration:

1. To display the Edit Business Components Configuration dialog, do one of the following:
 - In the Application Navigator, right-click the application module and choose **Configurations**. In the Manage Configurations dialog, select the default configuration named `appModuleNameLocal` and click **Edit**.
 - In the overview editor for the application module, click the **Configurations** tab and double-click the default configuration named `appModuleNameLocal` in the displayed list.
2. In the Edit Business Components Configuration dialog, edit the desired runtime properties and click **OK** to save the changes for your configuration.

9.3.4 How to Change the Database Connection for Your Project

When you are developing applications, you may have a number of different users or schemas that you want to switch between. You can do this by changing the connection properties of the project that contains the business components. The selection you make will automatically update the connection name for each configuration that your project's `bc4j.xcfg` file defines.

To change the connection used by your application module's configuration:

1. In the Application Navigator, double-click the application project.

2. In the Project Properties dialog, select **Business Components** to display the Business Components page, which shows details of the current database connection.
3. Click **Edit**, and in the Edit Database Connection dialog, make the appropriate changes.
4. Click **OK**.

9.3.5 What You May Need to Know About Application Module Connections

When testing your application module with the Business Component Browser, you should be aware of the connection configuration.

The Business Component Browser is an extremely useful tool for testing your application module's data model interactively. However, as a standalone Java tool, the Browser cannot run within the context of a Java EE application server. Therefore, you will not be able to test application modules using a configuration that depends on a JDBC data source. The solution is simply to test the application module by selecting a configuration that uses a JDBC URL connection. When you run the Business Component Browser, select the desired connection from the **Business Component Configuration Name** dropdown list in the displayed Select Business Components Configuration dialog.

9.4 Defining Nested Application Modules

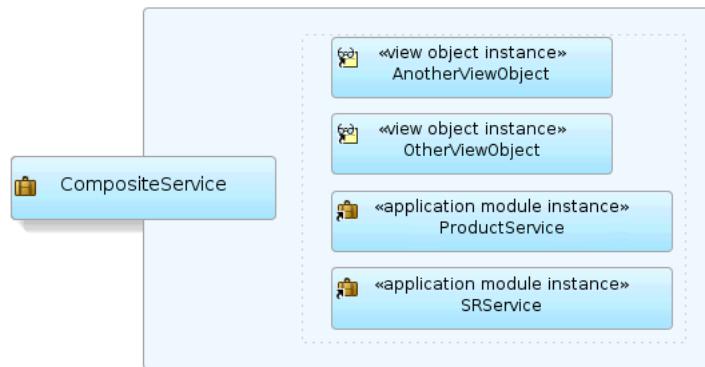
Application modules support the ability to create software components that mimic the modularity of your use cases, for which your higher-level functions might reuse a "subfunction" that is common to several business work flows. You can implement this modularity by defining composite application modules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

Declarative support for defining nested application modules is available through the overview editor for the application module, as shown in [Figure 9–13](#). The API for application modules also supports nesting of application modules at runtime.

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

Using the basic logic that an application module represents an end-user use case or work flow, you can build application modules that cater to the data required by some shared, modular use case, and then reuse those application modules inside of other more complicated application modules that are designed to support a more complex use case. For example, imagine that after creating the application modules `StoreServiceAM` and `ProductService`, you later need to build an application that uses both of these services as an integral part of a new `CompositeService` application module. [Figure 9–12](#) illustrates what this `CompositeService` would look like in a JDeveloper business components diagram. Notice that an application module like `CompositeService` can contain a combination of view object instances and application module instances.

Figure 9–12 Application Module Instances Can Be Reused to Assemble Composite Services



9.4.1 How to Define a Nested Application Module

To specify a composite root application module that nests an instance of an existing application module, use the overview editor for the application module. All of the nested component instances (contained by the application module instance) share the same transaction and entity object caches as the root application module that reuses an instance of them.

Tip: If you leverage nested application modules in your application, be sure to read [Section 11.2.1.4, "How Nested Application Modules Appear in the Data Controls Panel"](#) to avoid common pitfalls when performing data binding involving them.

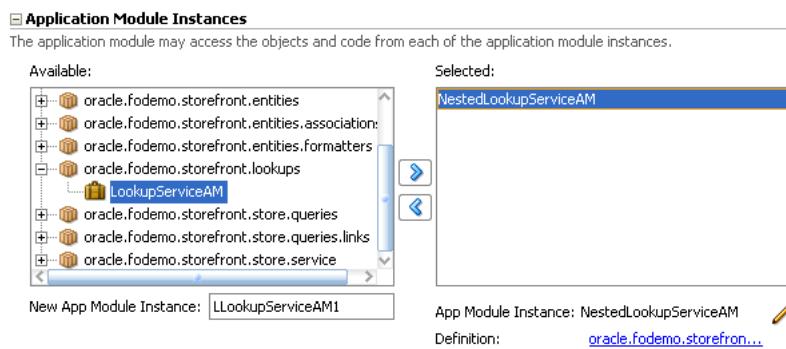
To define a nested application module:

1. In the Application Navigator, double-click the root application module.
2. In overview editor navigation list, click the **Data Model** navigation tab, and expand the **Application Module Instances** section.
3. To add a nested application module to the data model, first select it in the **Available** list.

The **New App Module Instance** field below the list shows the name that will be used to identify the nested application module that you add to the data model.

4. To change the name before adding it, type a different name in the **New App Module Instance** field.
5. With the desired application module selected, shuttle the application module to the **Selected** list.

[Figure 9–13](#) shows the application module `LookupServiceAM` has been renamed to `NestedLookupServiceAM` before it was shuttled to the **Selected** list.

Figure 9–13 Data Model Displays Added Application Module Instances

9.4.2 What You May Need to Know About Root Application Modules Versus Nested Application Module Usages

At runtime, your application works with a *main* — or what's known as a *root* — application module. Any application module can be used as a root application module; however, in practice the application modules that are used as root application modules are the ones that map to more complex end-user use cases, assuming you're not just building a straightforward CRUD application. When a root application module contains other nested application modules, they all participate in the root application module's transaction and share the same database connection and a single set of entity caches. This sharing is handled for you automatically by the root application module and its *Transaction* object.

Additionally, when you construct an application using an ADF bounded task flow, to declaratively manage the transactional boundaries, the ADF framework will automatically nest application modules used by the task flow at runtime. For details about bounded task flows and transactions, see [Section 17.2, "Managing Transactions"](#).

9.5 Creating an Application Module Diagram for Your Business Service

As you develop the business service's data model, it is often convenient to be able to visualize it using a UML model. JDeveloper supports easily creating a diagram for your application module that other developers can use for reference.

9.5.1 How to Create an Application Module Diagram

To create an application module diagram, use the Create Business Components Diagram dialog, which is available in the New Gallery.

To create a diagram of your application module:

1. In the Application Navigator, right-click the project in which you want to create the diagram and choose **New**.
2. In the New Gallery, expand **Business Tier**, select **ADF Business Components**, then select **Business Components Diagram**, and click **OK**.
3. In the Create Business Components dialog, enter a diagram name and a package name in which the diagram will be created.
4. Click **OK** to create the empty diagram and open the diagrammer.

5. To add your existing application module to the open diagram, select the desired application module in the Application Navigator and drop it onto the diagram surface.
6. Use the Property Inspector to:
 - Hide the package name
 - Change the font
 - Turn off the grid and page breaks
 - Turn off the display of the end names on the view link connectors ("Master"/"Detail")

After completing these steps, the diagram looks similar to the diagram shown in [Figure 9–14](#).

Figure 9–14 Partial UML Diagram of Application Module



9.5.2 What Happens When You Create an Application Module Diagram

When you create a business components diagram, JDeveloper creates a `.adfbc_diagram` file to represents the diagram in a subdirectory of the project's model path that matches the package name in which the diagram resides.

By default, the Application Navigator unifies the display of the project content's paths so that ADF components and Java files in the source path appear in the same package tree as the UML model artifacts in the project model path. You can use the **Navigator Display Options > Show Directories** toolbar option in the Application Navigator to switch between the unified directory view and a more distinct directory path view of the project content.

9.5.3 What You May Need to Know About Application Module Diagrams

You can do a number of tasks directly on the diagram, such as editing the application module, controlling display options, filtering methods names, showing related objects and files, publishing the application, and launching the Business Component Browser.

9.5.3.1 Using the Diagram for Editing the Application Module

The UML diagram of business components is not just a static picture that reflects the point in time when you dropped the application module onto the diagram. Rather, it is a UML-based rendering of the current component definitions, so it will always reflect the current state of affairs. The UML diagram is both a visualization aid and a visual navigation and editing tool. You can bring up the overview editor for any application module in a diagram by choosing **Properties** from the context menu (or by double-clicking the application module). You can also perform some application module editing tasks directly on the diagram, tasks such as renaming view object instances, dropping view object definitions onto the data model to create a new view object instance, and removing view object instances by pressing the Delete key.

9.5.3.2 Controlling Display Options

After selecting the application module in the diagram, use the Property Inspector to control its display options. In the **Display Options** category, toggle properties like the following:

- **Show Operations** — to display service methods
- **Show Package** — to display the package name
- **Show Stereotype** — to display the type of object (for example "<>application module<>")

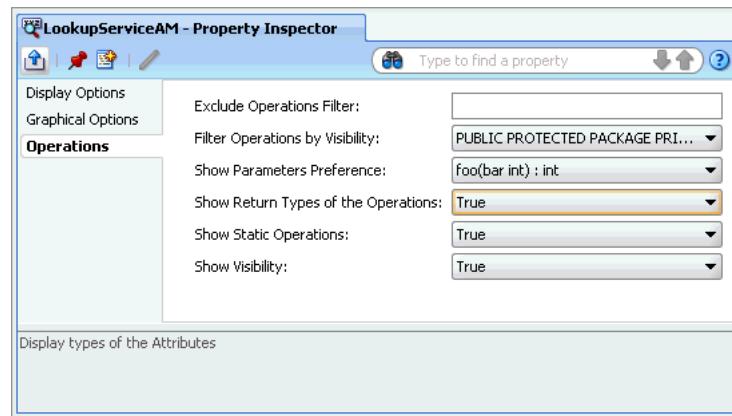
Note: The term *operation* is a more generic, UML name for methods.

In the **Operations** category, you will typically consider changing the following properties depending on the amount of detail you want to provide in the diagram:

- **Show Parameters Preference**
- **Show Return Types of the Operations**
- **Show Visibility** (public, private, etc.)

By default, all operations of the application module are fully displayed, as shown by the Property Inspector settings in [Figure 9–15](#).

Figure 9–15 Property Inspector with Default Diagrammer Options



On the context menu, you can also select to **View As**:

- **Symbolic** — to show service operations
- **Compact** — to show only the icon and the name
- **Expanded** — to show operations and data model (*default*)

9.5.3.3 Filtering Method Names

Initially, if you show the operations for the application module, the diagram displays all the methods. Any method it recognizes as an overridden framework method displays in the <>Framework<> operations category. The rest display in the <>Business<> methods category.

The **Exclude Operations Filter** property is a regular expression that you can use to filter out methods you don't want to display on the diagram. For example, by setting the **Exclude Operations Filter** property to:

```
findLoggedInUser.*|retrieveOrder.*|get.*
```

you can filter out all of the following application module methods:

- `findLoggedInUserByEmail`
- `retrieveOrderById`
- All the generated view object getter methods

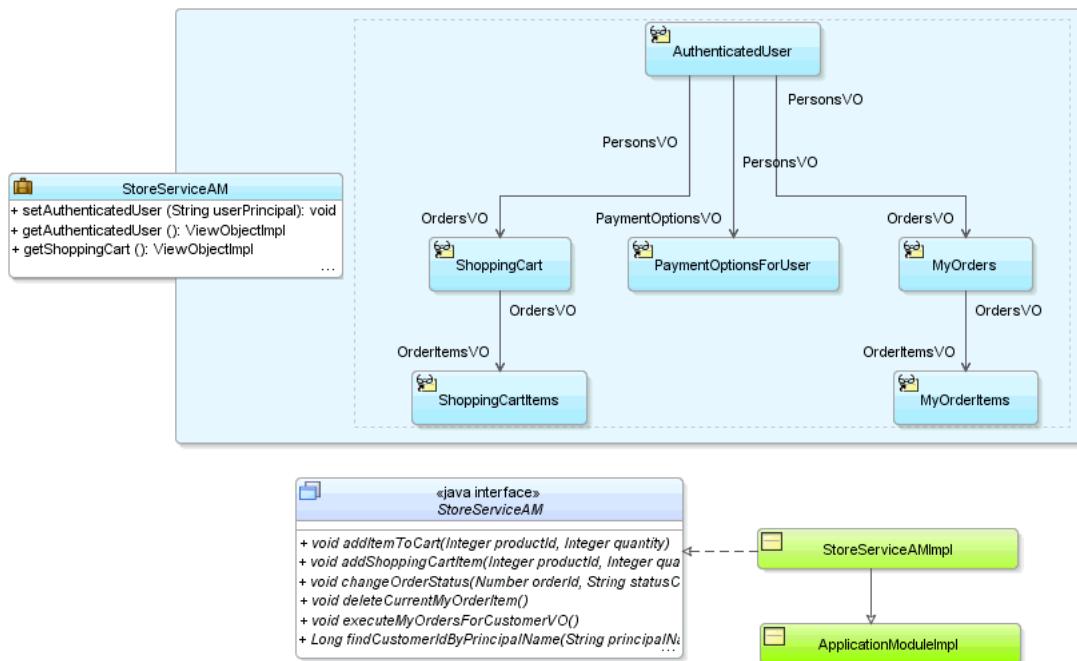
9.5.3.4 Showing Related Objects and Implementation Files

After selecting the application module on the diagram — or any set of individual view object instances in its data model — you can choose **Show > Related Elements** from the context menu to display related component definitions on the diagram. In a similar fashion, choosing **Show > Implementation Files** will include the files that implement the application module on the diagram. You can repeat these options on the additional diagram elements that appear until the diagram includes the level of detail you want to convey.

Note: Deleting components from the diagram only removes their visual representation on the diagram surface. The components and classes remain on the file system and in the Application Navigator.

[Figure 9–16](#) illustrates how the diagram displays the implementation files for an application module. You will see the related elements for the application module's implementation class (`StoreServiceAMImpl`). The diagram also draws an additional dependency line between the application module and the implementation class. If you have cast the application module instance to a specific custom interface, the diagram will also show that.

Figure 9–16 Adding Detail to a Diagram Using Show Related Elements and Show Implementation Files



9.5.3.5 Publishing the Application Module Diagram

To publish the diagram to PNG, JPG, SVG, or compressed SVG format, choose **Publish Diagram** from the context menu on the diagram surface.

9.5.3.6 Testing the Application Module from the Diagram

To launch the Business Component Browser for an application module in the diagram, choose **Run** from the context menu.

9.6 Supporting Multipage Units of Work

While interacting with your Fusion web application, end users might:

- Visit the same pages multiple times, expecting fast response times
- Perform a logical unit of work that requires visiting many different pages to complete
- Need to perform a partial "rollback" of a pending set of changes they've made but haven't saved yet.
- Unwittingly be the victim of an application server failure in a server farm before saving pending changes

The application module pooling and state management features simplify implementing scalable, well-performing applications to address these requirements.

Note: ADF bounded task flows can represent a transactional unit of work. You can specify options on the task flow to determine how to handle the transaction. For details about the declarative capabilities of ADF bounded task flows, see [Section 17.2, "Managing Transactions"](#).

9.6.1 How to Simulate State Management in the Business Component Browser

To simulate what the state management functionality does, you can launch two instances of Business Component Browser on an application module in the Application Navigator.

To simulate transaction state passivation using the tester:

1. Run the Business Component Browser and double-click a view object instance to query its data.
2. Make a note of the current values of a several attributes for a few rows.
3. Update those rows to have a different value those attributes, but do not commit the changes.
4. Choose **File > Save Transaction State** from the Business Component Browser main menu.

A Passivated Transaction State dialog appears, indicating a numerical transaction ID number. Make a note of this number.

5. Exit out of the Business Component Browser completely.
6. Restart the Business Component Browser and double-click the same view object instance to query its data.
7. Notice that the data is *not* changed. The queried data from the data reflects the current state of the database without your changes.

8. Choose **File > Restore Transaction State** from the Business Component Browser main menu, and enter the transaction ID you noted in Step 4.

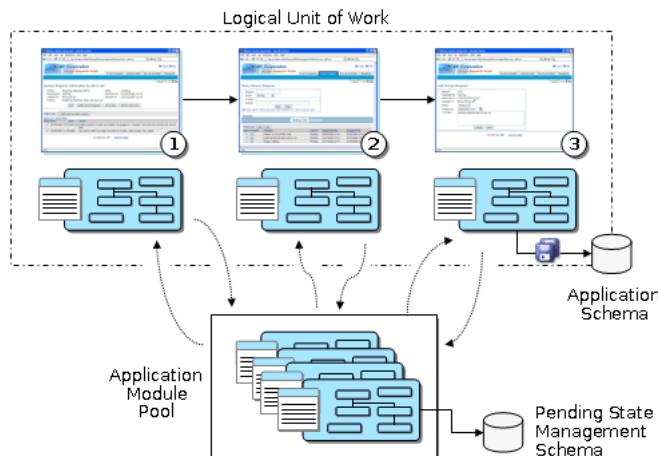
At this point, you'll see that your pending changes are reflected again in the rows you modified. If you commit the transaction now, your changes are permanently saved to the database.

9.6.2 What Happens When the Application Uses Application Module Pooling and State Management

Applications you build that leverage an application module as their business service take advantage of an automatic application module pooling feature. This facility manages a configurable set of application module instances that grows and shrinks as the end-user load on your application changes during the day. Due to the natural "think time" inherent in the end user's interaction with your application user interface, the number of application module instances in the pool can be smaller than the overall number of active users using the system.

As shown in [Figure 9–17](#), as a given end user visits multiple pages in your application to accomplish a logical task, with each page request an application module instance in the pool is acquired automatically from the pool for the lifetime of that one request. At the end of the request, the instance is automatically returned to the pool for use by another user session. In order to protect the end user's work against application server failure, the application module supports the ability to freeze the set of pending changes in its entity caches to a persistent store by saving an XML snapshot describing the change set. For scalability reasons, this state snapshot is typically saved in a state management schema that is a different database schema than the one containing the application data.

Figure 9–17 Using Pooled Application Modules Throughout a Multipage, Logical Unit of Work



The pooling algorithm affords a tunable optimization whereby a certain number of application module instances will attempt to stay "sticky" to the last user session that returned them to the pool. The optimization is not a guarantee, but when a user can benefit from the optimization, they continue to work with the same application module instance from the pool as long as system load allows. When load is too high, the pooling algorithm uses any available instance in the pool to service the user's request and the frozen snapshot of their logical unit of work is reconstituted from the persistent store to allow the new instance of the application module to continue where

the last one left off. The end user continues to work in this way until they commit or roll back their changes.

Using these facilities, the application module delivers the productivity of a stateful development paradigm that can easily handle multipage work flows, in an architecture that delivers the runtime performance near that of a completely stateless application. You will learn more about these application module features in [Chapter 36, "Application State Management"](#) and about how to tune them in [Chapter 37, "Understanding Application Module Pooling"](#).

Note: This application module pooling and state management is also available for thin-client, desktop-fidelity Swing applications and web-style user interfaces.

9.7 Customizing an Application Module with Service Methods

An application module can expose its data model of view objects to clients without requiring any custom Java code. This allows client code to use the `ApplicationModule`, `ViewObject`, `RowSet`, and `Row` interfaces in the `oracle.jbo` package to work directly with any view object in the data model. However, just because you *can* programmatically manipulate view objects any way you want to in client code doesn't mean that doing so is always a best practice.

Whenever the programmatic code that manipulates view objects is a logical aspect of implementing your complete business service functionality, you should encapsulate the details by writing a custom method in your application module's Java class. This includes code that:

- Configures view object properties to query the correct data to display
- Iterates over view object rows to return an aggregate calculation
- Performs any kind of multistep procedural logic with one or more view objects

By centralizing these implementation details in your application module, you gain the following benefits:

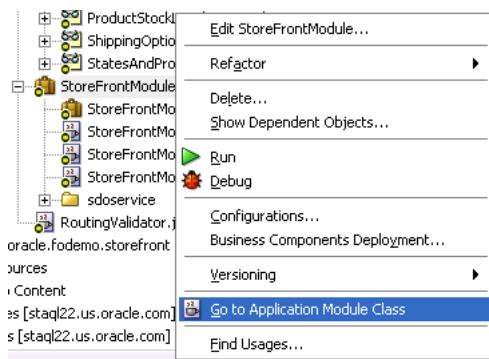
- You make the intent of your code more clear to clients.
- You allow multiple client pages to easily call the same code if needed.
- You simplify regression-testing of your complete business service functionality.
- You keep the option open to improve your implementation without affecting clients.
- You enable *declarative* invocation of logical business functionality in your pages.

9.7.1 How to Generate a Custom Class for an Application Module

To add a custom service method to your application module, you must first enable a custom Java class for it. If you have configured your IDE-level Business Components Java generation preferences to automatically generate an application module class, a custom class will be present. If you're not sure whether your application module has a custom Java class, open the overview editor for the application module node in the Application Navigator. The Java Classes page of the editor displays the complete list of classes generated for the application module in the project. If the file exists because someone created it already, then the Java Classes page will display a linked file name identified as the **Application Module Class**. To open an existing file in the source editor, click on the corresponding file name link.

You can also check the application module node's context menu for the **Go to Application Module Class** option, as shown in [Figure 9–18](#). When this option is present in the menu, you can use it to quickly navigate to your application module's custom class. If you don't see the option in the menu, then your application module is currently an XML-only component.

Figure 9–18 Quickly Navigating to an Application Module's Custom Java Class



If no Java class exists in your project, you can generate one using the Java Classes page of the overview editor for the application module.

To generate a Java file for your application module class:

1. In the Application Navigator, double-click the application module.
2. In the overview editor, click the **Java** navigation tab.
3. On the Java Classes page, click **Edit Java Options**.
4. In the Select Java Options dialog, select **Generate Application Module Class**.
5. Click **OK**.

The new .java file will appear in the Java Classes page.

9.7.2 What Happens When You Generate a Custom Class for an Application Module

When you generate a custom class for an application module, JDeveloper creates the file in the same directory as the component's XML component definition file. The default name for its custom Java file will be `AppModuleClassNameImpl.java`.

The Java generation option choices you made for the application module persist on the Java Classes page on subsequent visits to the overview editor for the application module. Just as with the XML definition file, JDeveloper keeps the generated code in your custom Java classes up to date with any changes you make in the editor. If later you decide you do not require a custom Java file, from the Java Classes page open the Select Java Options dialog and deselect **Generate Application Module Class** to remove the custom Java file from the project.

9.7.3 What You May Need to Know About Default Code Generation

By default, the application module Java class will look similar to what you see in [Example 9–5](#) when you've first enabled it. Of interest, it contains:

- Getter methods for each view object instance in the data model
- A `main()` method allowing you to debug the application module using the Business Component Browser

Example 9–5 Default Application Module Generated Code

```

package devguide.model;
import devguide.model.common.StoreServiceAM;
import oracle.jbo.server.ApplicationModuleImpl;
import oracle.jbo.server.ViewLinkImpl;
import oracle.jbo.server.ViewObjectImpl;
// -----
// --- File generated by Oracle ADF Business Components Design Time.
// --- Custom code may be added to this class.
// --- Warning: Do not modify method signatures of generated methods.
// -----
public class StoreServiceAMImpl extends ApplicationModuleImpl {
    /** This is the default constructor (do not remove) */
    public StoreServiceImpl() { }

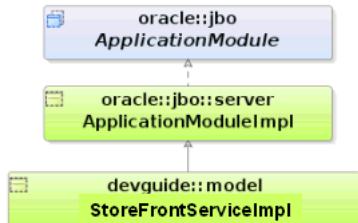
    /** Container's getter for YourViewObjectInstance1 */
    public ViewObjectImpl getYourViewObjectInstance1() {
        return (ViewObjectImpl)findViewObject("YourViewObjectInstance1");
    }

    // ... Additional ViewObjectImpl getters for each view object instance
    // ... ViewLink getters for view link instances here
}

```

As shown in [Figure 9–19](#), your application module class extends the base ADF `ApplicationModuleImpl` class to inherit all the default behavior before adding your custom code.

Figure 9–19 Your Custom Application Module Class Extends `ApplicationModuleImpl`



9.7.4 How to Add a Custom Service Method to an Application Module

To add a custom service method to an application module, simply navigate to the application module's custom class and enter the Java code for a new method into the application module's Java implementation class. Use the following guidelines to decide on the appropriate visibility for the method:

- If you will use the method only inside this component's implementation as a helper method, make the method `private`.
- If you want to allow eventual subclasses of your application module to be able to invoke or override the method, make it `protected`.
- If you need clients to be able to invoke it, it must be `public`.

Note: The `StoreServiceAM` application module examples in this chapter use the strongly typed, custom entity object classes that you saw illustrated in the `StoreServiceAMImpl2.java` example at the end of Chapter 4, "Creating a Business Domain Layer Using Entity Objects".

[Example 9–6](#) shows a `private retrieveServiceRequestById()` helper method in the `StoreServiceAMImpl.java` class for the `StoreServiceAM` application module. It uses the `static getDefinition()` method of the `ServiceRequestImpl` entity object class to access its related entity definition, it uses the `createPrimaryKey()` method on the entity object class to create an appropriate Key object to look up the service request, and then it uses the `findByPrimaryKey()` method on the entity definition to find the entity row in the entity cache. It returns an instance of the strongly typed `ServiceRequestImpl` class, the custom Java class for the `ServiceRequest` entity object.

Example 9–6 Private Helper Method in Custom Application Module Class

```
// In devguide.model.StoreServiceAMImpl class
/*
 * Helper method to return a ServiceRequest by Id
 */
private ServiceRequestImpl retrieveServiceRequestById(long requestId) {
    EntityDefImpl svcReqDef = ServiceRequestImpl.getDefinitionObject();
    Key svcReqKey =
        ServiceRequestImpl.createPrimaryKey(new DBSequence(requestId));
    return (ServiceRequestImpl)svcReqDef.findByPrimaryKey(getDBTransaction(),
        svcReqKey);
}
```

[Example 9–7](#) shows a `public createProduct()` method that allows the caller to pass in a name and description of a product to be created. It uses the `getDefinition()` method of the `ProductImpl` entity object class to access its related entity definition, and it uses the `createInstance2()` method to create a new `ProductImpl` entity row, whose `Name` and `Description` attributes it populates with the parameter values passed in before committing the transaction.

Example 9–7 Public Method in Custom Application Module Class

```
/*
 * Create a new Product and Return its new id
 */
public long createProduct(String name, String description) {
    EntityDefImpl productDef = ProductImpl.getDefinitionObject();
    ProductImpl newProduct =
        (ProductImpl)productDef.createInstance2(getDBTransaction(),null);
    newProduct.setName(name);
    newProduct.setDescription(description);
    try {
        getDBTransaction().commit();
    }
    catch (JboException ex) {
        getDBTransaction().rollback();
        throw ex;
    }
    DBSequence newIdAssigned = newProduct.getProdId();
    return newIdAssigned.getSequenceNumber().longValue();
```

```
}
```

9.7.5 How to Test the Custom Application Module Using a Static Main Method

When you are ready to test the methods of your custom application module, you can use JDeveloper to generate JUnit test cases. With JUnit, you can use any of the programmatic APIs available in the `oracle.jbo` package to work with the application module and invoke the custom methods. For details about using JUnit with ADF Business Components, see [Section 29.8, "Regression Testing with JUnit"](#).

As an alternative to JUnit test cases, a common technique to test your custom application module methods is to write a simple test case. For example, you could build the testing code into an object and include that code in the `static main()` method. [Example 9–8](#) shows a sample `main()` method you could add to your custom application module class to test the sample methods you will write. You'll make use of a Configuration object (see [Section 6.4.2, "How to Create a Command-Line Java Test Client"](#)) to instantiate and work with the application module for testing.

Note: The fact that this Configuration object resides in the `oracle.jbo.client` package suggests that it is used for accessing an application module as an application *client*. Because a `main()` method is a kind of programmatic, command-line client, so this is an acceptable practice. Furthermore, even though you typically would not cast the return value of `createRootApplicationModule()` directly to an application module's implementation class, it is legal to do so in this one situation since despite being a client to the application module, the `main()` method's code resides right inside the application module implementation class itself.

A quick glance through the code in [Example 9–8](#) shows that it exercises the four methods created in the previous examples to:

1. Retrieves the status of service request 101.
2. Retrieves the name of the technician assigned to service request 101.
3. Sets the status of service request 101 to the illegal value "Reopened".
4. Creates a new product supplying a null product name.
5. Creates a new product and display its newly assigned product ID.

Example 9–8 Sample Main Method to Test a Custom Application Module from the Inside

```
// Main method in StoreServiceAMImpl.java
public static void main(String[] args) {
    String      amDef = "devguide.model.StoreServiceAM";
    String      config = "StoreServiceAMLocal";
    ApplicationModule am =
        Configuration.createRootApplicationModule(amDef,config);
    /*
     * NOTE: This cast to use the StoreServiceAMImpl class is OK since this
     * ----- code is inside a business tier *Impl.java file and not in a
     *       client class that is accessing the business tier from "outside".
     */
    StoreServiceAMImpl service = (StoreServiceAMImpl)am;
    // 1. Retrieve the status of service request 101
    String status = service.findServiceRequestStatus(101);
    System.out.println("Status of SR# 101 = " + status);
```

```
// 2. Retrieve the name of the technician assigned to service request 101
String techName = service.findServiceRequestTechnician(101);
System.out.println("Technician for SR# 101 = " + techName);
try {
    // 3. Set the status of service request 101 to illegal value "Reopened"
    service.updateRequestStatus(101, "Reopened");
}
catch (JboException ex) {
    System.out.println("ERROR: "+ex.getMessage());
}
long id = 0;
try {
    // 4. Create a new product supplying a null product name
    id = service.createProduct(null, "Makes Blended Fruit Drinks");
}
catch (JboException ex) {
    System.out.println("ERROR: "+ex.getMessage());
}
// 5. Create a new product and display its newly assigned product id
id = service.createProduct("Smoothie Maker", "Makes Blended Fruit Drinks");
System.out.println("New product created successfully with id = "+id);
Configuration.releaseRootApplicationModule(am,true);
}
```

Running the custom application module class calls the `main()` method in [Example 9–8](#), and shows the following output:

```
Status of SR# 101 = Closed
Technician for SR# 101 = Bruce Ernst
ERROR: The status must be Open, Pending, or Closed
ERROR: JBO-27014: Attribute Name in Product is required
New product created successfully with id = 209
```

Notice that the attempt to set the service request status to "Reopened" failed because the List Validator failed on the `ServiceRequest` entity object's `Status` attribute. That validator was configured to allow only values from the static list `Open`, `Pending`, or `Closed`. Also notice that the first attempt to call `createProduct()` with a null for the product name raises an exception due to the built-in mandatory validation on the `Name` attribute of the `Product` entity object.

Note: For an explanation of how you can use the client application to invoke the custom service methods that you create in your custom application module, see [Section 9.8, "Publishing Custom Service Methods to UI Clients"](#).

9.8 Publishing Custom Service Methods to UI Clients

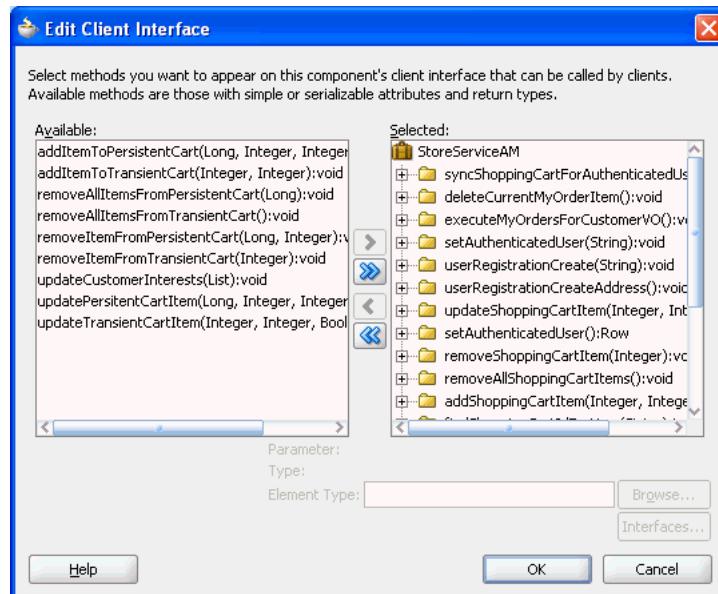
When you add a public custom method to your application module class, if you want your application's UI to be able to invoke it, you need to include the method on the application module's UI client interface.

9.8.1 How to Publish a Custom Method on the Application Module's Client Interface

To include a public method from your application module's custom Java class on the client interface, use the Java Classes page of the overview editor for the application module, and then click the **Edit** icon in the **Client Interface** section of the page to display the Edit Client Interface dialog. Select one or more desired methods from the

Available list and click the **Add** button to shuttle them into the **Selected** list. Then click **OK** to close the editor. [Figure 9–20](#) shows multiple public methods added to the client interface.

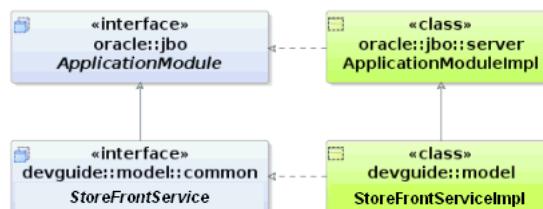
Figure 9–20 Public Methods Added to an Application Module's Client Interface



9.8.2 What Happens When You Publish Custom Service Methods

When you publish custom service methods on the client interface, as shown in [Figure 9–21](#), JDeveloper creates a Java interface with the same name as the application module in the common subpackage of the package in which your application module resides. For an application module named `StoreServiceAM` in the `fodemo.model` package, this interface will be named `StoreServiceAM` and reside in the `fodemo.model.common` package. The interface extends the base `ApplicationModule` interface in the `oracle.jbo` package, reflecting that a client can access all of the base functionality that your application module inherits from the `ApplicationModuleImpl` class.

Figure 9–21 Custom Client Interface Extends the Base ApplicationModule Interface



As shown in [Example 9–9](#), the `StoreServiceAM` interface includes the method signatures of all of the methods you've selected to be on the client interface of your application module.

Example 9–9 Custom Client Interface Based on Methods Selected in the Client Interface Panel

```
package fodemo.model.common;
```

```

import oracle.jbo.ApplicationModule;
// -----
// --- File generated by Oracle ADF Business Components Design Time.
// -----
public interface StoreServiceAM extends ApplicationModule {
    long createProduct(String name, String description);
    String findServiceRequestStatus(long requestId);
    String findServiceRequestTechnician(long requestId);
    void updateRequestStatus(long requestId, String newStatus);
}

```

Each time you add or remove methods in the **Client Interface** section, the corresponding client interface file is updated automatically. JDeveloper also generates a companion client proxy class that is used when you deploy your application module for access by a remote client. For the StoreServiceAM application module in this example, the client proxy file is called `StoreServiceAMClient` and it is created in the `devguide.model.client` subpackage.

Note: After adding new custom methods to the client interface, if your new custom methods do not appear to be available when you use JDeveloper's code insight context-sensitive statement completion, try recompiling the generated client interface. To do this, select the application module in the Application Navigator, select the source file for the interface of the same name in the Structure window, and choose **Rebuild** from the context menu. Consider this tip for new custom methods added to view objects and view rows as well.

9.8.3 How to Generate Client Interfaces for View Objects and View Rows

In addition to generating a client interface for your application module, it is also possible to generate strongly typed client interfaces for working with the other key client objects that you can customize. For example, you can open Java page in the overview editor for a view object, you can then expand the **Client Interface** section and the **Client Row Interface** section and add custom methods to the view object client interface and the view row client interface, respectively.

If for the Products view object in the `devguide.model.queries` package you were to enable the generation of a custom view object Java class and add one or more custom methods to the view object client interface, JDeveloper would generate the `ProductsImpl` class and `Products` interface, as shown in [Figure 9–22](#). As with the application module custom interface, notice that it gets generated in the common subpackage.

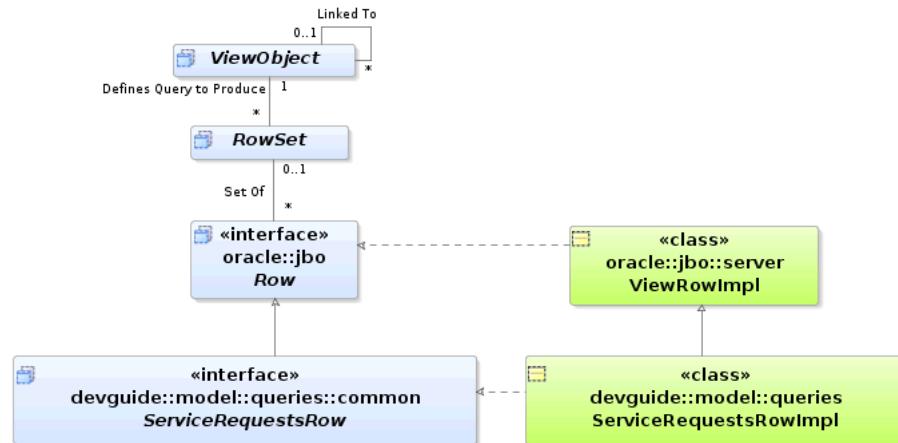
Figure 9–22 Custom View Object Interface Extends the Base ViewObject Interface



Likewise, if for the same view object you were to enable the generation of a custom view row Java class and add one or more custom methods to the view row client

interface, JDeveloper would generate the `ProductsRowImpl` class and `ProductsRow` interface, as shown in [Figure 9–23](#).

Figure 9–23 Custom View Row Interface Extends the Base Row Interface



9.8.4 What You May Need to Know About Method Signatures on the Client Interface

You can include any custom method in the client interface that obeys these implementation rules:

- If the method has a non-`void` return type, the type must be serializable.
- If the method accepts any parameters, all their types must be serializable.
- If the method signature includes a `throws` clause, the exception must be an instance of `JboException` in the `oracle.jbo` package.

In other words, all the types in its method signature must implement the `java.io.Serializable` interface, and any checked exceptions must be `JboException` or its subclass. Your method can throw any unchecked exception — `java.lang.RuntimeException` or a subclass of it — without disqualifying the method from appearing on the application module's client interface.

Note: If the method you've added to the application module class doesn't appear in the **Available** list, first verify that it doesn't violate any of the method implementation rules. If it seems like it should be a legal method, try recompiling the application module class before visiting the overview editor for the application module again.

9.8.5 What You May Need to Know About Passing Information from the Data Model

The private implementation of an application module custom method can easily refer to any view object instance in the data model using the generated accessor methods. By calling the `getCurrentRow()` method on any view object, it can access the same current row for any view object that the client user interface sees as the current row. As a result, while writing application module business service methods, you may not need to pass in parameters from the client. This is true if you would be passing in values only from the current rows of other view object instances in the same application module's data model.

For example, the custom application module method in [Example 9–10](#) accepts no parameters. Internally, the `createServiceRequest()` method calls

`getGlobals().getCurrentRow()` to access the current row of the `Globals` view object instance. Then it uses the strongly typed accessor methods on the row to access the values of the `ProblemDescription` and `ProductId` attributes to set them as the values of corresponding attributes in a newly created `ServiceRequest` entity object row.

Example 9–10 Using View Object Accessor Methods to Access a Current Row

```
// In StoreServiceAMImpl.java, createServiceRequest() method
GlobalsRowImpl globalsRow = (GlobalsRowImpl) getGlobals().getCurrentRow();
newReq.setProblemDescription(globalsRow.getProblemDescription());
newReq.setProdId(globalsRow.getProductId());
```

9.9 Debugging the Application Module Using the Business Component Browser

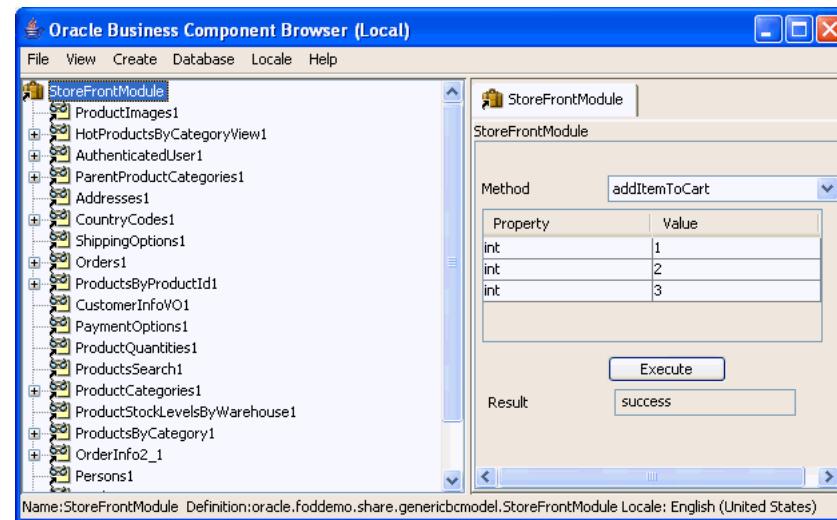
You can exercise your application module under the JDeveloper debugger while using the Business Component Browser as the testing interface. To debug an application module using the Business Component Browser, you can use either of these techniques:

- In the Application Navigator, right-click the application module and choose **Debug** from the context menu.
- In the source editor for an open application module XML or Java file, right-click inside the editor and select **Debug** from the context menu.

You can test the methods of your custom application module in the Business Component Browser after you have published them on the client interface, as described in [Section 9.8, "Publishing Custom Service Methods to UI Clients"](#).

Double-click the application module node in the tester window to open the methods testing panel. Select the desired method from the dropdown list and enter values to pass as method parameters. Click **Execute** and view the return value (if any) and test result. The result displayed in the tester will indicate whether or not the method executed successfully, as shown in [Figure 9–24](#).

Figure 9–24 Tester Displays Custom Application Module Method Results



9.10 Working Programmatically with an Application Module's Client Interface

After publishing methods on your application module's client interface, you can invoke those methods from a client.

9.10.1 How to Work Programmatically with an Application Module's Client Interface

To work programmatically with an application module's client interface, do the following:

- Cast `ApplicationModule` to the more specific client interface.
- Call any method on the interface.

Note: For simplicity, this section focuses on working only with the custom application module interface; however, the same downcasting approach works on the client to use a `ViewObject` interface as a view object interface like `ServiceRequests` or a `Row` interface as a custom view row interface like `ServiceRequestsRow`.

[Example 9–11](#) illustrates a `TestClientCustomInterface` class that puts these two steps into practice. You could also use the `main()` method of this class to test application module methods, as described in [Section 9.7.5, "How to Test the Custom Application Module Using a Static Main Method"](#). Here you use it to call all of the same methods from the client using the `StoreServiceAM` client interface.

The basic logic of the example follows these steps:

1. Acquire the application module instance and cast it to the more specific `StoreServiceAM` client interface.

Note: If you work with your application module using the default `ApplicationModule` interface in the `oracle.jbo` package, you won't have access to your custom methods. Make sure to cast the application module instance to your more specific custom interface like the `StoreServiceAM` interface in this example.

2. Call `findRequestStatus()` to find the status of service request 101.
3. Call `findServiceRequestTechnician()` to find the name of the technician assigned to service request 101.
4. Call `updateRequestStatus()` to try updating the status of request 101 to the illegal value `Reopened`.
5. Call `createProduct()` to try creating a product with a missing product name attribute value, and display the new product ID assigned to it.

Example 9–11 Using the Custom Interface of an Application Module from the Client

```
package devguide.client;
import devguide.model.common.StoreServiceAM;
import oracle.jbo.JboException;
import oracle.jbo.client.Configuration;
public class TestClientCustomInterface {
    public static void main(String[] args) {
```

```
String      amDef = "devguide.model.StoreServiceAM";
String      config = "StoreServiceAMLocal";
/*
 * This is the correct way to use application custom methods
 * from the client, by using the application module's automatically-
 * maintained custom service interface.
 */
// 1. Acquire instance of application module, cast to client interface
StoreServiceAM service =
    (StoreServiceAM)Configuration.createRootApplicationModule(amDef,config);
// 2. Find the status of service request 101
String status = service.findServiceRequestStatus(101);
System.out.println("Status of SR# 101 = " + status);
// 3. Find the name of the technician assigned to service request 101
String techName = service.findServiceRequestTechnician(101);
System.out.println("Technician for SR# 101 = " + techName);
try {
    // 4. Try updating the status of service request 101 to an illegal value
    service.updateRequestStatus(101,"Reopened");
}
catch (JboException ex) {
    System.out.println("ERROR: "+ex.getMessage());
}
long id = 0;
try {
    // 5. Try creating a new product with a missing required attribute
    id = service.createProduct(null,"Makes Blended Fruit Drinks");
}
catch (JboException ex) {
    System.out.println("ERROR: "+ex.getMessage());
}
// 6. Try creating a new product with a missing required attribute
id = service.createProduct("Smoothie Maker","Makes Blended Fruit Drinks");
System.out.println("New product created successfully with id = "+id);
Configuration.releaseRootApplicationModule(service,true);
}
}
```

9.10.2 What Happens When You Work with an Application Module's Client Interface

If the client layer accessing your application module will be located in the same tier of the Java EE architecture, the application module will be deployed in what is known as *local mode*. In local mode, the client interface is implemented directly by your custom application module Java class. Typically, you access an application module in local mode when you need to:

- Access the application module in the web tier of a JavaServer Faces application
- Access the application module in the web tier of a JSP/Struts application
- Access the application module in the client tier (two-tier client-server style) for a Swing application

In contrast, when the client layer accessing your application module is located in a *different* tier of the Java EE architecture, the application module will be deployed in what is known as *remote mode*. In remote mode, the generated client proxy class implements your application module client interface on the client side, and the class handles all of the communications details of working with the remotely deployed application module service. You typically access the application module in remote

mode only when a thin-client Swing application needs to access the application module on a remote application server.

A unique feature of ADF Business Components is that by adhering to the interface-only approach for working with client service methods, you can be sure your client code works unchanged regardless of your chosen deployment mode. Even if you plan to work only in local mode, the Java EE development community favors the interface-based approach to working with services. Using application modules, it's extremely easy to follow this approach in your applications.

Note: Whether you plan to deploy your application modules in local mode or remote mode, as described in [Section 9.8.4, "What You May Need to Know About Method Signatures on the Client Interface"](#), the JDeveloper design time ensures that your custom interface methods will use serializable types. This allows you to switch at any time between local mode or remote mode, or to support both at the same time, with no code changes.

9.10.3 How to Access an Application Module Client Interface in a Fusion Web Application

The Configuration class in the `oracle.jbo.client` package makes it very easy to get an instance of an application module for *testing*. This eases writing test client programs like the test client program described in [Section 29.8, "Regression Testing with JUnit"](#) as part of the JUnit regression testing fixture.

Because it is easy, it is tempting for developers to use the class `createRootApplicationModule()` and `releaseApplicationModule()` methods *anywhere* to access an application module. However, for Fusion web applications you should resist this temptation because there is an even easier way.

When working with Fusion web applications using the ADF Model layer for data binding, JDeveloper configures a servlet filter in your user interface project called the `ADFBindingFilter`. It orchestrates the automatic acquisition and release of an appropriate application module instance based on declarative binding metadata, and ensures that the service is available to be looked up as a data control using a known action binding or iterator binding, specified by any page definition file in the user interface project. You may eventually want to read about the ADF binding container, data controls, page definition files, and bindings, as described in [Chapter 11, "Using Oracle ADF Model in a Fusion Web Application"](#). For now, it is enough to realize that you can access the application module's client interface from this `DCBindingContainer` by naming an ADF action binding or an ADF iterator binding. You can reference the binding context and call methods on the custom client interface in a JSF managed bean, as shown in [Example 9–12](#) for an action binding and [Example 9–13](#) for an iterator binding.

To access the custom interface of your application module using an action binding, follow these basic steps (as illustrated in [Example 9–12](#)):

1. Access the ADF binding container.
2. Find a named action binding. (Use the name of any available action binding in the page definition files of the user interface project.)
3. Get the data control by name from the action binding.
4. Access the application module data provider from the data control.
5. Cast the application module to its client interface.

6. Call any method on the client interface.

Example 9–12 Accessing the Application Module Client Interface in a JSF Backing Bean Using a Named Action Binding

```
package demo.view;
import oracle.fodemo.storefront.store.service.common.StoreServiceAM;
import oracle.adf.model.binding.DCBindingContainer;
import oracle.adf.model.binding.DCDataControl;
import oracle.jbo.ApplicationModule;
import oracle.jbo.uicli.binding.JUCtrlActionBinding;
public class YourBackingBean {
    public String commandButton_action() {
        // Example using an action binding to get the data control
        public String commandButton_action() {
            // 1. Access the binding container
            DCBindingContainer bc = (DCBindingContainer)getBindings();
            // 2. Find a named action binding
            JUCtrlActionBinding action =
                (JUCtrlActionBinding)bc.findCtrlBinding("SomeActionBinding");
            // 3. Get the data control from the iterator binding (or method binding)
            DCDataControl dc = action.getDataControl();
            // 4. Access the data control's application module data provider
            ApplicationModule am = (ApplicationModule)dc.getDataProvider();
            // 5. Cast the AM to call methods on the custom client interface
            StoreServiceAM service = (StoreServiceAM)am;
            // 6. Call a method on the client interface
            service.doSomethingInteresting();
            return "SomeNavigationRule";
        }
    }
}
```

To access the custom interface of your application module using an iterator binding, follow these basic steps (as illustrated in [Example 9–13](#)):

1. Access the ADF binding container.
2. Find a named iterator binding. (Use the name of any iterator binding in the page definition files of the user interface project.)
3. Get the data control by name from the iterator binding.
4. Access the application module data provider from the data control.
5. Cast the application module to its client interface.
6. Call any method on the client interface.

Example 9–13 Accessing the Application Module Client Interface in a JSF Backing Bean Using a Named Iterator Binding

```
package demo.view;
import oracle.fodemo.storefront.store.service.common.StoreServiceAM;
import oracle.adf.model.binding.DCBindingContainer;
import oracle.adf.model.binding.DCDataControl;
import oracle.adf.model.binding.DCIteratorBinding;
import oracle.jbo.ApplicationModule;
public class YourBackingBean {
    public String commandButton_action() {
        // Example using an iterator binding to get the data control
        public String commandButton_action() {
            // 1. Access the binding container
            DCBindingContainer bc = (DCBindingContainer)getBindings();
```

```

        // 2. Find a named iterator binding
        DCIteratorBinding iter = bc.findIteratorBinding("SomeIteratorBinding");
        // 3. Get the data control from the iterator binding
        DCDataControl dc = iter.getDataControl();
        // 4. Access the data control's application module data provider
        ApplicationModule am = (ApplicationModule)dc.getDataProvider();
        // 5. Cast the AM to call methods on the custom client interface
        StoreServiceAM service = (StoreServiceAM)am;
        // 6. Call a method on the client interface
        service.doSomethingInteresting();
        return "SomeNavigationRule";
    }
}

```

These backing bean examples depend on the helper method shown in [Example 9–14](#).

Example 9–14 Helper Method for Backing Bean Class

```

public BindingContainer getBindings() {
    if (this.bindings == null) {
        FacesContext fc = FacesContext.getCurrentInstance();
        this.bindings =
            (BindingContainer)fc.getApplication().evaluateExpressionGet(fc,
                "#{bindings}", BindingContainer.class);
    }
    return this.bindings;
}

```

If you create the backing bean class by overriding a button that is declaratively bound to an ADF action, then JDeveloper will automatically generate this method in your class. Otherwise, you will need to add the helper method to your class yourself.

9.11 Overriding Built-in Framework Methods

The `ApplicationModuleImpl` base class provides a number of built-in methods that implement its functionality. While [Appendix E, "Most Commonly Used ADF Business Components Methods"](#) provides a quick reference to the most common code that you will typically write, use, and override in your custom application module classes, this section focuses on helping you understand the basic steps to override one of these built-in framework methods to augment the default behavior.

9.11.1 How to Override a Built-in Framework Method

To override a built-in framework method for an application module, use the **Override Methods** dialog, which you select for the application module Java class from the main menu.

To override an application module framework method:

1. In the Application Navigator, double-click the application module.
2. In the overview editor, select the **Java** navigation tab.
3. On the Java Classes page, click the linked file name of the application module Java class that you want to customize. JDeveloper opens the class file in the source editor.
4. From the JDeveloper toolbar, choose **Source > Override Methods**.

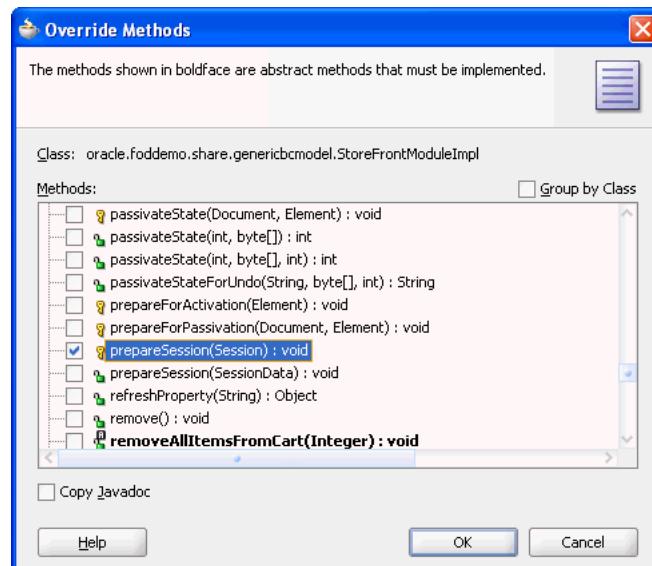
If the **Source** menu is not displayed in the JDeveloper toolbar, be sure the desired Java class file is open and the source editor is visible.

5. In the Override Methods dialog, scroll the list to locate the desired methods or type the first few letters of the method name to perform an incremental search.
6. Select one or more methods.

The Override Methods dialog allows you to select any number of methods to override simultaneously.

For example, if you wanted to override the application module's `prepareSession()` method to augment the default functionality when a new user session begins working with an application module service component for the first time, you would select the checkbox next to the `prepareSession(Session)` method, as shown in [Figure 9–25](#).

Figure 9–25 Overriding a Built-in Framework Method



7. Click OK.

9.11.2 What Happens When You Override a Built-in Framework Method

When you dismiss the Override Methods dialog, you return to the source editor with the cursor focus on the overridden method, as shown in [Figure 9–26](#). Notice that the method appears with a single line that calls `super.prepareSession()`. This is the syntax in Java for invoking the default behavior that the base class would have normally performed for this method. By adding code before or after this line in the custom application module class, you can augment the default behavior before or after the default functionality.

Figure 9–26 Source Editor Margin Gives Visual Feedback About Overridden Methods

```

316  @Override
317  protected void prepareSession(Session session) {
318      super.prepareSession(session);
319  }

```

Also notice that when you override a method using the Override Methods dialog, the source editor inserts the JDK `@Override` annotation just before the overridden method. This causes the compiler to generate a compile-time error if the method in the

application module class does not match the signature of any method in the superclass.

Be careful when you add method names to your class to override a method in the superclass; you must have the signature *exactly* the same as the base class method you want to override. Be sure to add the @Override annotation just before the method. This way, if your method does not match the signature of any method in the superclass, the compiler will generate a compile-time error. Also, when you write code for a method instead of calling the superclass implementation, you should have a thorough understanding of what built-in code you are suppressing or replacing.

9.11.3 How to Override `prepareSession()` to Set Up an Application Module for a New User Session

Since the `prepareSession()` method is invoked by the application module when it is used for the first time by a new user session, it's a useful method to override in your custom application module class to perform setup tasks that are specific to each new user that uses your application module. [Example 9-15](#) illustrates an overridden `prepareSession()` method in the `devguide.model.StoreServiceAMImpl` class that invokes a `findLoggedInUserByEmailInStaffList()` helper method to initialize the `StaffList` view object instance to display the row corresponding to the currently logged-in user.

The helper method does the following:

1. Calls `super.prepareSession()` to perform the default processing
2. Accesses the `StaffList` view object instance using the generated `getStaffList()` getter method
3. Calls the `getUserPrincipalName()` method to get name of the currently authenticated user

Note: The `getUserPrincipalName()` API in the sample below will return null instead of the authenticated user's name. So, the example contains the fallback code that assigns a fixed email ID of `sking` for testing purposes. For more information about securing your application, see [Chapter 28, "Adding Security to a Fusion Web Application"](#).

4. Defines a `CurrentUser` named bind variable, with the `currentUser` member variable as its default value
5. Sets an additional WHERE clause to find the current user's row by email
6. Executes the query to retrieve the `StaffList` row for the current user

After overriding the `prepareSession()` method in this way, if you test the `StoreServiceAM` application module using the Business Component Browser, you'll see that the `StaffList` view object instance has the single row corresponding to Steven King (`email = 'sking'`).

Example 9-15 Initializing the StaffList View Object Instance to Display a Current User's Information

```
// In devguide.model.StoreServiceAMImpl class
protected void prepareSession(Session session) {
    // 1. Call the superclass to perform the default processing
    super.prepareSession(session);
```

```
        findLoggedInUserByEmailInStaffList();
    }
    private void findLoggedInUserByEmailInStaffList() {
        // 2. Access the StaffList vo instance using the generated getter method
        ViewObject staffList = getStaffList();
        // 3. Get the name of the currently authenticated user
        String currentUserName = getUserPrincipalName();
        /*
         * Until later when we learn how to integrate Java_EE security,
         * this API will return null. For testing, we can default it
         * to the email of one of the staff members like "sking".
         */
        if (currentUserName == null) {
            currentUserName = "sking";
        }
        /*
         * We can't use a simple findByKey since the key for the
         * StaffList view object is the numerical userid of the staff member
         * and we want to find the user by their email address. We could build
         * an "EMAIL = :CurrentUser" where clause directly into the view object
         * at design time, but here let's illustrate doing it dynamically to
         * see this alternative.
         */
        // 4. Define named bind variable, with currentUserName as default value
        staffList.defineNamedWhereClauseParam("CurrentUser",      // bindvar name
                                              currentUserName, // default value
                                              null);
        // 5. Set an additional WHERE clause to find the current user's row by email
        staffList.setWhereClause("EMAIL = :CurrentUser");
        // 6. Execute the query to retrieve the StaffList row for the current user
        staffList.executeQuery();
        /*
         * If the view object needs to be also used during this session
         * without the additional where clause, you would use
         * setWhereClause(null) and removeNamedWhereClauseParam("CurrentUser") to
         * leave the view object instance back in its original state.
         */
    }
}
```

Sharing Application Module View Instances

This chapter describes how to organize your business services project to most efficiently utilize read-only data accessed from lookup tables or other static data source, such as a flat file.

This chapter includes the following sections:

- [Section 10.1, "Introduction to Shared Application Modules"](#)
- [Section 10.2, "Sharing an Application Module Instance"](#)
- [Section 10.3, "Defining a Base View Object for Use with Lookup Tables"](#)
- [Section 10.4, "Accessing View Instances of the Shared Service"](#)
- [Section 10.5, "Testing View Object Instances in a Shared Application Module"](#)

10.1 Introduction to Shared Application Modules

Web applications often utilize data that is required across sessions and does not change very frequently. An example of this type of *static data* might be displayed in the application user interface in a lookup list. Each time your application accesses the static data, you could incur an unnecessary overhead when the static data caches are repopulated from the database for each application session on every request. In order to optimize performance, a common practice when working with ADF Business Components is to cache the shared static data for reuse across sessions and requests.

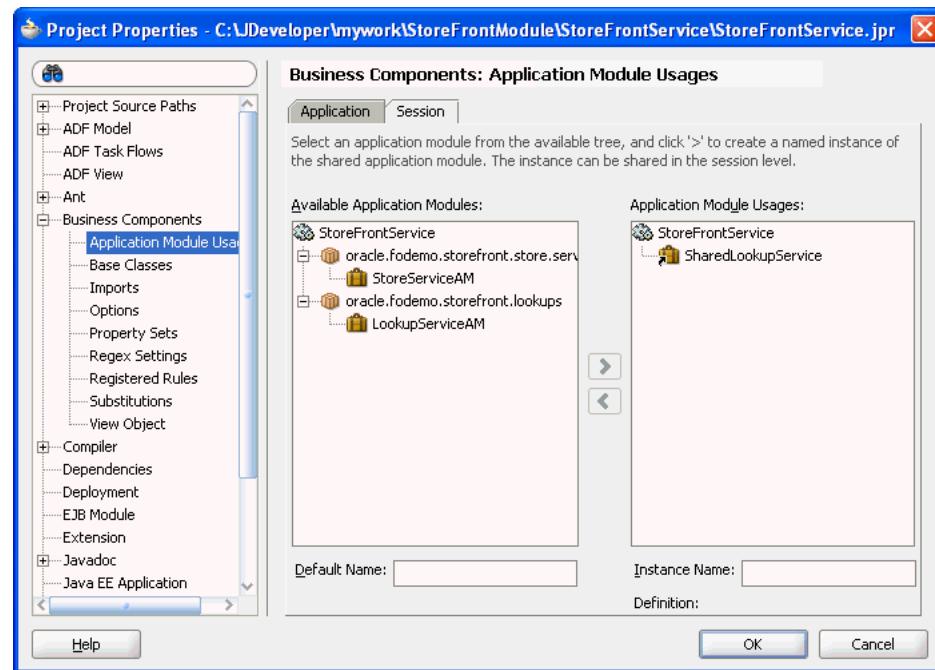
10.2 Sharing an Application Module Instance

Declarative support for shared data caches is available in JDeveloper through the Project Properties dialog. Creating a shared application module allows requests from multiple sessions to share a single application module instance which is managed by an application pool for the lifetime of the web server virtual machine.

Best Practice: Use a shared application module to group view instances when you want to reuse lists of static data across the application. The shared application module can be configured to allow any user session to access the data or it can be configured to restrict access to just the UI components of a single user session. For example, you can use a shared application module to group view instances that access lookup data, such as a list of countries. The use of a shared application module allows all shared resources to be managed in a single place and does not require a scoped managed bean for this purpose.

As shown in [Figure 10–1](#), the Project Properties dialog lets you specify application-level or session-level sharing of the application module's data model. In the case of *application-level sharing*, any user session will be able to access the same view instances contained in the shared application module. In contrast, the lifecycle of the *session-level shared* application module extends to a user session. In this case, the view instances of the application module will be accessible by other UI components in the same user session.

Figure 10–1 Project Properties Dialog Defines Shared Application Module Instance



When you create the data model for the application module that you intend to share, be sure that the data in cached row sets will not need to be changed either at the application level or session level. For example, in the application-level shared application module, view instances should query only static data such as state codes or currency types. If a view object instance queries data that depends on the current user, then the query can be cached at the session level and shared by all components that reference the row-set cache. For example, the session-level shared application module might contain a view instance with data security that takes a manager as the current user to return the list of direct reports. In this case, the cache of direct reports would exist for the duration of the manager's session.

10.2.1 How to Create a Shared Application Module Instance

To create a shared application module instance, use the Project Properties dialog. You define a logical name for a distinct, separate root application module that will hold your application's read-only data.

To create a shared application module instance:

1. In the Application Navigator, right-click the project in which you want to create the shared application module and choose **Project Properties**.
2. In the Project Properties dialog, expand **Business Components** and select **Application Module Usage**.

3. On the Business Components: Application Module Usage page, select one of these tabs:
 - When you want to define the shared application module for the context of the application, select the **Application** tab.
 - When you want to define the shared application module for the context of the current user session, select the **Session** tab
4. In the **Available Application Modules** list, select the desired application module and shuttle it to the **Application Module Usages** list.
5. Assign the application module a unique instance name.
 The shared application module instance (of either scope) must have a unique instance name. Supplying a meaningful name will also help to clarify which shared application module instance a given usage is referencing.
6. Click **OK**.

10.2.2 What Happens When You Define a Shared Application Module

JDeveloper automatically creates the `AppModuleShared` configuration when you create an application module. The presence of this configuration in the `bc4j.xcfg` file informs JDeveloper that the application module is a candidate to be shared, and allows JDeveloper to display the application module in the **Available Application Modules** list of the Project Properties dialog's Application Module Usage page.

The `AppModuleShared` configuration sets these properties on the application module to enable sharing and help to maintain efficient use of the shared resource at runtime:

- `jbo.ampool.isuseexclusive` is set to `false` to specify that requests from multiple sessions can share a single instance of the application module, which is managed by the application pool for the lifetime of the web server virtual machine. When you do not enable application module sharing, JDeveloper sets the value `true` to repopulate the data caches from the database for each application session on every request.
- `jbo.ampool.maxpoolsize` is set to 1 (one) to specify that only a single application module instance will be created for the ADF Business Components application module pool. This setting enforces the efficient use of the shared application module resource and prevents unneeded multiple instances of the shared application module from being created at runtime.

You can view the shared application module's configuration by choosing **Configurations** from the context menu on the application module in the Application Navigator. JDeveloper saves the `bc4j.xcfg` file in the `./common` subdirectory relative to the application module's XML component definition. If you remove the configuration or modify the values of the `jbo.ampool` runtime properties (`isuseexclusive`, `maxpoolsize`), the application module will not be available to use as a shared application module instance.

For example, if you look at the `bc4j.xcfg` file in the `./src/oracle/fodemo/storefront/lookups/common` directory of the Fusion Order Demo application's `StoreFrontService` project, you will see the two named configurations for the `LookupServiceAM` application module, as shown in [Example 10-1](#). Specifically, the `LookupServiceAMShared` configuration sets the `jbo.ampool` runtime properties on the shared application module instance. For more information about the ADF Business Components application module pooling and

runtime configuration of application modules, see [Chapter 10, "Sharing Application Module View Instances"](#).

Example 10–1 *LookupServiceAMShared Configuration in the bc4j.xcfg File*

```
<BC4JConfig version="11.0" xmlns="http://xmlns.oracle.com/bc4j/configuration">
  < AppModuleConfigBag>
    < AppModuleConfig name="LookupServiceAMLocal" DeployPlatform="LOCAL"
      JDBCName="FOD"
      ApplicationName="oracle.fodemo.storefront.lookups.LookupServiceAM"
      jbo.project="StoreFrontService"
      java.naming.factory.initial="oracle.jbo.common.JboInitialContextFactory">
      <Security>
        AppModuleJndiName="oracle.fodemo.storefront.lookups.LookupServiceAM"/>
      </Security>
    </ AppModuleConfig>
    < AppModuleConfig name="LookupServiceAMShared"
      DeployPlatform="LOCAL" JDBCName="FOD"
      ApplicationName="oracle.fodemo.storefront.lookups.LookupServiceAM"
      jbo.project="StoreFrontService"
      java.naming.factory.initial="oracle.jbo.common.JboInitialContextFactory">
      <AM-Pooling jbo.ampool.isuseexclusive="false" jbo.ampool.maxpoolsize="1"/>
      <Security>
        AppModuleJndiName="oracle.fodemo.storefront.lookups.LookupServiceAM"/>
      </Security>
    </ AppModuleConfig>
  </ AppModuleConfigBag>
  < ConnectionDefinition name="FOD">
    <Entry value="1521" name="JDBC_PORT"/>
    <Entry value="JDBC" name="ConnectionType"/>
    <Entry value="localhost" name="HOSTNAME"/>
    <Entry value="FOD" name="user"/>
    <Entry value="FOD" name="ConnectionName"/>
    <Entry value="XE" name="SID"/>
    <Entry value="thin" name="ORACLE_JDBC_TYPE"/>
    <SecureEntry name="password">
      <![CDATA[{904}05F1395DAAFCBCD4A4A0149A7AB6B1DB71]]>
    </SecureEntry>
    <Entry name="DeployPassword" value="true"/>
  </ ConnectionDefinition>
</BC4JConfig>
```

Because the shared application module can be accessed by any Business Components project in the same application workspace, JDeveloper maintains the scope of the shared application module in the Business Components project configuration file (.jpx). This file is saved in the `src` directory of the project. For example, if you look at the `StoreFrontService.jpx` file in the `./src` directory of the Fusion Order Demo application's `StoreFrontService` project, you will see that the `SharedLookupService` application module's usage definition specifies `SharedScope = 2`, corresponding to application-level sharing, as shown in [Example 10–2](#). An application module that you set to session-level sharing will show `SharedScope = 1`.

Example 10–2 *Application Module Usage Configuration in the .jpx File*

```
<JboProject
  xmlns="http://xmlns.oracle.com/bc4j"
  Name="StoreFrontService"
  SeparateXMLFiles="true"
  PackageName="">
  . . .
< AppModuleUsage
```

```

Name="SharedLookupService"
FullName="oracle.fodemo.storefront.lookup.LookupServiceAM"
ConfigurationName="oracle.fodemo.storefront.lookup.LookupServiceAMShared"
SharedScope="2" />
</JboProject>

```

10.2.3 What You May Need to Know About Shared Application Module Instances

When you create a Business Components project to access the data model of the shared application module, several restrictions apply.

10.2.3.1 Design Time Scope of the Shared Application Module

Defining the shared application module in the Project Properties dialog makes the application module's data model available to other Business Components projects of the same application workspace only. When you want to make the data model available beyond the application workspace, you can publish the data model as an ADF Library, as described in [Chapter 31, "Reusing Application Components"](#).

When viewing a data control usage from the `DataBinding.cpx` file in the Structure window, do not set the **Configuration** property to a shared application module configuration. By default, for an application module named `AppModuleName`, the Property Inspector will list the configurations named `AppModuleNameShared` and `AppModuleNameLocal`. At runtime, Oracle ADF uses the shared configuration automatically when you configure an application as a shared application module, but the configuration is not designed to be used by an application module data control usage. For more information about data control usage, see [Chapter 11.4, "Working with the DataBindings.cpx File"](#).

10.2.3.2 Design Time Scope of View Instances of the Shared Application Module

In JDeveloper, you must define view accessors on the business component definition for the project that will permit access to view instances of the shared application module. The view accessor lets you point from an entity object or view object definition in one Business Components project to a view object definition or view instance in a shared application module. For details about creating view accessors for this purpose, see [Section 10.4, "Accessing View Instances of the Shared Service"](#).

10.3 Defining a Base View Object for Use with Lookup Tables

When your application needs to display static data, you can define a shared application module with view instances that most likely will access lookup tables. A lookup table is a static, translated list of data to which the application refers. Lookup table data can be organized in the database in various ways. While it is possible to store related lookup data in separate tables, it is often convenient to combine all of the lookup information for your application within a single table. For example, a column `LOOKUP_TYPE` created for the `ORDERS_LOOKUP` table would serve to partition one table that might contain diverse codes such as `FWK_TBX_YES_NO` for the values yes and no, `FWK_TBX_COUNTRY` for country names, and `FWK_TBK_CURRENCY` for the names of national currencies.

When your database schema organizes lookup data in a single database table, you want to avoid creating individual queries for each set of data. Instead, you will use the overview editor to define a single, base view object that maps the desired columns of the lookup table to the view object attributes you define. Since only the value of the `LOOKUP_TYPE` column will need to change in the query statement, you can add view

criteria on the view object definition to specify a WHERE clause that will set the LOOKUP_TYPE value. In this way, your application encapsulates access to the lookup table data in a single view object definition that will be easy to maintain when a LOOKUP_TYPE value changes or your application needs to query additional lookup types.

10.3.1 How to Create a Base View Object Definition for a Lookup Table

The base view object that queries columns of the lookup table will be a read-only view object, since you do not need to handle updating data or require any of the benefits provided by entity-based view objects. (For a description of those benefits, see [Section 5.1.2, "Runtime Features Unique to Entity-Based View Objects"](#).)

Note: While read-only view objects you create to access lookup tables are ideal for inclusion in a shared application module, if you intend to share the view object in a shared application module instance, you must create the view object in the same package as the shared application module.

To create a read-only view object, use the Create View Object wizard, which is available from the New Gallery.

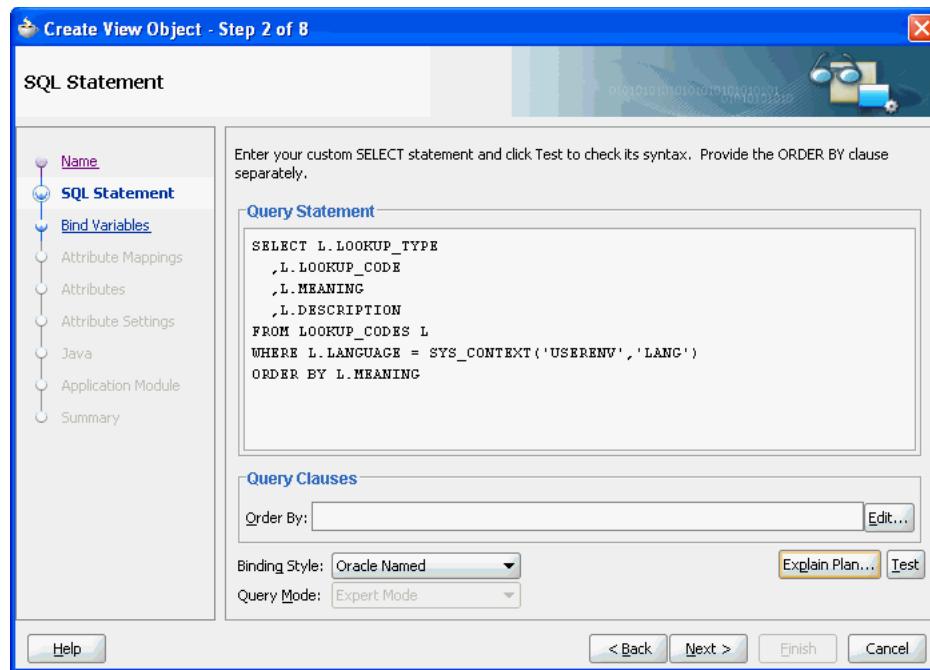
To create a base view object for a lookup table:

1. In the Application Navigator, locate the shared application module you created in which you want to create the view object, right-click its package node, and choose **New**.
2. In the New Gallery, expand **Business Tier** and select **ADF Business Components**.
3. In the Items list, select **View Object**.
4. In the Create View Object wizard, on the **Name** page, enter a package name and a view object name.

When naming the package, consider creating a separate package for the lookup.

5. Select **Read-only Access** to indicate that you want this view object to manage data with read-only access and click **Next**.
6. On the SQL Statement page, enter your SQL statement directly into the **Query Statement** box.

Your query names the columns of the lookup table, similar to the SQL statement shown in [Figure 10–2](#) to query the LOOKUP_CODE, MEANING, and DESCRIPTION columns in the LOOKUP_CODES table.

Figure 10–2 Create View Object Wizard, SQL Query for Lookup Table

7. After entering the query statement, no other changes are required. Click **Next**.
8. On the Bind Variables page, click **Next**.
9. On the Attribute Mappings page, note the mapped view object attribute names displayed and click **Next**.

By default, the wizard creates Java-friendly view object attribute names that correspond to the SELECT list column names.

10. On the Attribute Settings page, you can optionally rename the attributes to use any names that might be more appropriate. Choose the attribute to rename from the **Select Attributes** dropdown list. When you are finished, click **Next**.

For example, the Fusion Order Demo application renames the default attributes `LookupType` and `LookupCode` to `Type` and `Value` respectively. Changes you make to the view object definition will not change the underlying query.

11. On the Java page, click **Next**.
12. On the Application Module page, do not add an instance of the view object to the application module data model. Click **Finish**.

The shared application module data model will include view instances based on view criteria that you add to the base view object definition. In this way, you do not need to create an individual view object to query each `LOOKUP_TYPE` value. For details about adding the view object instances to the data model, see [Section 9.2.3, "How to Add a View Object to an Application Module"](#).

10.3.2 What Happens When You Create a Base View Object

When you create the view object definition for the lookup table, JDeveloper first describes the query to infer the following from the columns in the SELECT list:

- The Java-friendly view attribute names (for example, `LookupType` instead of `LOOKUP_TYPE`)

By default, the wizard creates Java-friendly view object attribute names that correspond to the SELECT list column names.

- The SQL and Java data types of each attribute

JDeveloper then creates the XML component definition file that represents the view objects's declarative settings and saves it in the directory that corresponds to the name of its package. For example, the XML file created for a view object named `LookupsBaseVO` in the `lookups` package is `./lookups/LookupsBaseVO.xml` under the project's source path.

To view the view object settings, expand the desired view object in the Application Navigator, select the XML file under the expanded view object, and open the Structure Window. The Structure window displays the list of definitions, including the SQL query and the properties of each attribute. To open the file in the editor, double-click the corresponding `.xml` node. As shown in [Example 10–3](#), the `LookupsBaseVO.xml` file defines one `<SQLQuery>` definition and one `<ViewAttribute>` definition for each mapped column. Without a view criteria to filter the query results, the view object query returns the `LOOKUP_CODE`, `LOOKUP_MEANING`, and `LOOKUP_DESCRIPTION` and maps them to view instance attribute values for Value, Name, and Description respectively.

Example 10–3 LookupsBaseVO SQL Query and Attribute Mapping Definition

```
<ViewObject
    xmlns="http://xmlns.oracle.com/bc4j"
    Name="LookupsBaseVO"
    BindingStyle="OracleName"
    CustomQuery="true"
    PageIterMode="Full"
    UseGlueCode="false">
    <SQLQuery>
        <! [CDATA[SELECT L.LOOKUP_TYPE
            ,L.LOOKUP_CODE
            ,L.MEANING
            ,L.DESCRIPTION
            FROM LOOKUP_CODES L
            WHERE L.LANGUAGE = SYS_CONTEXT('USERENV', 'LANG')
            ORDER BY L.MEANING] ]>
    </SQLQuery>
    <DesignTime>
        <Attr Name="_CodeGenFlag2" Value="Access|VarAccess"/>
        <Attr Name="_isExpertMode" Value="true"/>
        <Attr Name="_version" Value="11.1.1.44.61"/>
    </DesignTime>
    <ViewAttribute
        Name="Type"
        IsUpdateable="false"
        AttrLoad="Each"
        IsPersistent="false"
        IsNotNull="true"
        PrecisionRule="true"
        Precision="255"
        Type="java.lang.String"
        ColumnType="VARCHAR2"
        AliasName="LOOKUP_TYPE"
        Expression="LOOKUP_TYPE"
        SQLType="VARCHAR">
        <DesignTime>
            <Attr Name="_DisplaySize" Value="30"/>
        </DesignTime>
    </ViewAttribute>

```

```
</DesignTime>
</ViewAttribute>
<ViewAttribute
  Name="Value"
  IsUpdateable="false"
  AttrLoad="Each"
  IsPersistent="false"
  IsNotNull="true"
  PrecisionRule="true"
  Precision="30"
  Type="java.lang.String"
  ColumnType="VARCHAR2"
  AliasName="LOOKUP_CODE"
  Expression="LOOKUP_CODE"
  SQLType="VARCHAR">
  <DesignTime>
    <Attr Name="_DisplaySize" Value="30"/>
  </DesignTime>
</ViewAttribute>
<ViewAttribute
  Name="Name"
  IsUpdateable="false"
  AttrLoad="Each"
  IsPersistent="false"
  IsNotNull="true"
  PrecisionRule="true"
  Precision="80"
  Type="java.lang.String"
  ColumnType="VARCHAR2"
  AliasName="MEANING"
  Expression="MEANING"
  SQLType="VARCHAR">
  <DesignTime>
    <Attr Name="_DisplaySize" Value="80"/>
  </DesignTime>
</ViewAttribute>
<ViewAttribute
  Name="Description"
  IsUpdateable="false"
  AttrLoad="Each"
  IsPersistent="false"
  PrecisionRule="true"
  Precision="240"
  Type="java.lang.String"
  ColumnType="VARCHAR2"
  AliasName="DESCRIPTION"
  Passivate="true"
  Expression="DESCRIPTION"
  SQLType="VARCHAR">
  <DesignTime>
    <Attr Name="_DisplaySize" Value="240"/>
  </DesignTime>
</ViewAttribute>
  .
  .
</ViewObject>
```

10.3.3 How to Define the WHERE Clause of the Lookup View Object Using View Criteria

You create named view criteria definitions in the data model project when you need to filter view object results. View criteria that you define at design time can participate in UI scenarios that require filtering of data.

Use the Edit View Criteria dialog to create the view criteria definition for the lookup base view object you defined to query the lookup table. The editor lets you build a WHERE clause using attribute name instead of the target view object's corresponding SQL column names. The resulting definition will include:

- One view criteria row consisting of one view criteria group, with a single view criteria item used to define the lookup view object's Type attribute.
- The view criteria item will consist of an Type attribute name, the Equal operator, and the value of the LOOKUP_TYPE that will filter the query results.

Because a single view criteria is defined, no logical conjunctions are needed to bracket the WHERE clause conditions.

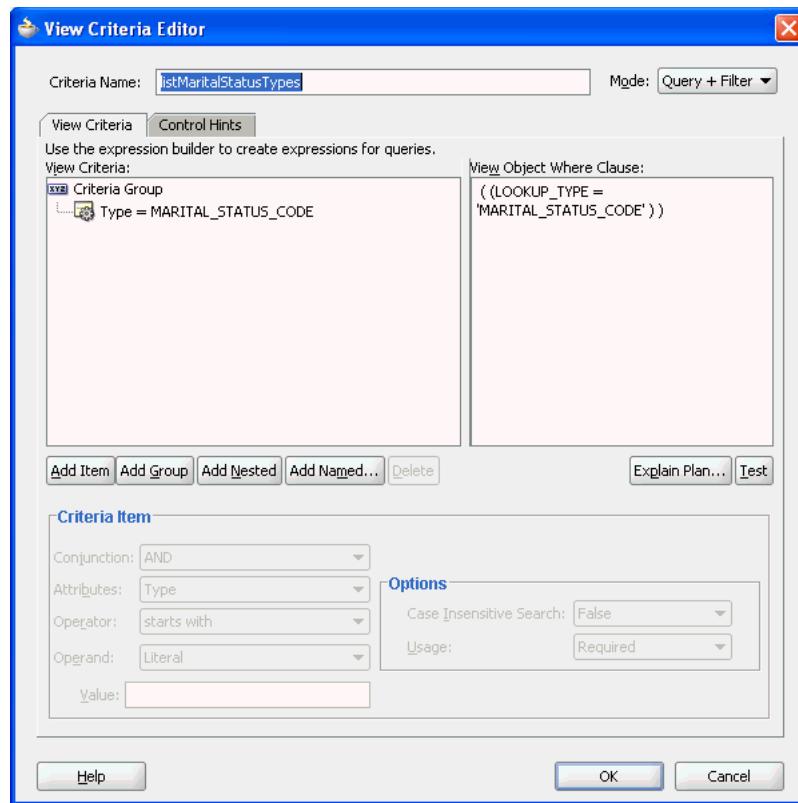
To create LOOKUP_TYPE view criteria for the lookup view object:

1. In the Application Navigator, double-click the lookup base view object you defined.
2. In the overview editor navigation list, select **Query**.
3. In the **View Criteria** section, click the **Create New View Criteria** icon.
4. In the Edit View Criteria dialog, on the View Criteria page, click the **Add Item** button to add a single criteria item to the view criteria group.
5. In the **Criteria Item** panel, define the criteria item as follows:
 - Choose **Type** as the attribute (or other name that you defined for the attribute the view object maps to the LOOKUP_TYPE column).
 - Choose **equal to** as the operator.
 - Keep **Literal** as the operand choice and enter the value name that defines the desired type. For example, to query the marital status codes, you might enter the value MARITAL_STATUS_CODE corresponding to the LOOKUP_TYPE column.

Leave all other settings unchanged.

The view object WHERE clause shown in the editor should display a simple criteria similar to the one shown in [Figure 10–3](#), where the value MARITAL_STATUS_CODE is set to filter the LOOKUP_TYPE column.

6. Click **OK**.
7. Repeat this procedure to define one view criteria for each LOOKUP_TYPE that you wish to query.

Figure 10–3 View Criteria Editor Dialog with Lookup View Object View Criteria Specified

10.3.4 What Happens When You Create a View Criteria with the Editor

The view criteria design time editor in JDeveloper lets you easily create view criteria and save them as named definitions. These named view criteria definitions add metadata to the target view object's own definition. Once defined, named view criteria appear by name in the Query page of the overview editor for the view object.

JDeveloper then creates the XML component definition file that represents the view objects's declarative settings and saves it in the directory that corresponds to the name of its package. For example, the XML file created for a view object named `LookupsBaseVO` in the `lookups` package is `./lookups/LookupsBaseVO.xml` under the project's source path.

To view the view criteria, expand the desired view object in the Application Navigator, select the XML file under the expanded view object, open the Structure window, and expand the View Criteria node. As shown in [Example 10–4](#), the `LookupsBaseVO.xml` file specifies the `<ViewCriteria>` definition that allows the `LookupsBaseVO` to return only the marital types. Other view criteria added to the `LookupsBaseVO` are omitted from this example for brevity.

Example 10–4 *listMaritalStatusTypes* View Criteria in the Lookup View Object Definition

```
<ViewObject
    xmlns="http://xmlns.oracle.com/bc4j"
    Name="LookupsBaseVO"
    BindingStyle="OracleName"
    CustomQuery="true"
    PageIterMode="Full"
    UseGlueCode="false">
```

```
<SQLQuery>
<! [CDATA[SELECT L.LOOKUP_TYPE
,L.LOOKUP_CODE
,L.MEANING
,L.DESCRIPTION
FROM LOOKUP_CODES L
WHERE L.LANGUAGE = SYS_CONTEXT('USERENV', 'LANG')
ORDER BY L.MEANING]]>
</SQLQuery>
...
<ViewCriteria
    Name="listMaritalStatusTypes"
    ViewObjectName="oracle.fodemo.storefront.lookups.LookupsBaseVO"
    Conjunction="AND"
    Mode="3"
    AppliedIfJoinSatisfied="false">
<Properties>
    <Property Name="autoExecute" Value="false"/>
    <Property Name="showInList" Value="true"/>
    <Property Name="mode" Value="Basic"/>
</Properties>
<ViewCriteriaRow
    Name="vcrow24">
    <ViewCriteriaItem
        Name="Type"
        ViewAttribute="Type"
        Operator="="
        Conjunction="AND"
        Value="MARITAL_STATUS_CODE"
        Required="Optional"/>
</ViewCriteriaRow>
</ViewCriteria>
```

10.3.5 What Happens at Runtime When a View Instance Accesses Lookup Data

When you create a view instance based on a view criteria, the next time the view instance is executed it augments its SQL query with an additional WHERE clause predicate corresponding to the view criteria that you've populated in the view criteria rows.

10.4 Accessing View Instances of the Shared Service

View accessors in ADF Business Components are value accessor objects that point from an entity object attribute (or view object) to a destination view object or shared view instance in the same application workspace. The view accessor returns a row set that by default contains all rows from the destination view object. You can optionally filter this row set by applying view criteria to the view accessor. The base entity object or view object on which you create the view accessor and the destination view object need not be in the same project or application module, but they must be in the same application workspace.

Because view accessors give you the flexibility to reach across application modules to access the queried data, they are ideally suited for accessing view instances of shared application modules. For details about creating a data model of view instances for a shared application module, see [Section 10.2.1, "How to Create a Shared Application Module Instance"](#).

This ability to access view objects in different application modules makes view accessors particularly useful for:

- Validation rules that you set on the attributes of an entity object. In this case, the view accessor derives the validation rule's values from lookup data corresponding to a view instance attribute in the shared application module.
- List of Value (LOV) that you enable for the attribute of any view object. In this case, the view accessor derives the list of values from lookup data corresponding to a view instance attribute in the shared application module.

Validation rules with accessors are useful when you do not want the UI to display a list of values to the user, but you still need to restrict the list of valid values.

Alternatively, consider defining an LOV for view object attributes to simplify the task of working with list controls in the user interface. Because you define the LOV on the individual attributes of business components, you can customize the LOV usage for an attribute once and expect to see the list control in the form wherever the attribute appears.

10.4.1 How to Create a View Accessor for an Entity Object or View Object

Entity-based view objects inherit view accessors that you define on their base entity objects. Thus, defining the view accessor once on the entity object itself allows you to reuse the same view accessor, whether you want to define validation rules for entity object attributes or to create LOV-enabled attributes for that entity object's view object. However, when you do not anticipate using view accessors for validation rules, you can add the view accessor directly to the view object that defines the LOV-enabled attribute.

For example, in the `StoreFrontModule` package of the Fusion Order Demo application, the `AddressEO` entity object defines the `Shared_CountriesVA` view accessor and the `AddressesVO` view object inherits this view accessor. In this case, defining the view accessor on the entity object is useful: the accessor for `AddressEO` defines a validation rule on the `CountryId` attribute and the same accessor for `AddressesVO` enables an LOV on its `CountryId` attribute.

When you create a view accessor that accesses a view instance from a shared application module, you may want to use a prefix like `Shared_` to name the view accessor. This naming convention will help you identify the view accessor when you need to select it for the entity object or view object.

You can further refine the list returned by a view accessor by applying view criteria that you define on the view object. To create view criteria for use with a view accessor, see [Section 10.3.3, "How to Define the WHERE Clause of the Lookup View Object Using View Criteria"](#).

To create the view accessor:

1. In the Application Navigator, double-click the entity object or view object on which you want to define the view accessor.
Whether you create the view accessor on the entity object or on the view object will depend on the view accessor's intended usage. Generally, creating view accessors on the entity object ensures the widest possible usage.
2. In the overview editor navigation list, select **View Accessors** and click the **Create New View Accessors** icon to add the accessor to the entity object or view object definition you are currently editing.

3. In the View Accessors dialog, select the view instance name you created for your lookup table from the shared application module node and shuttle it to the view accessors list.

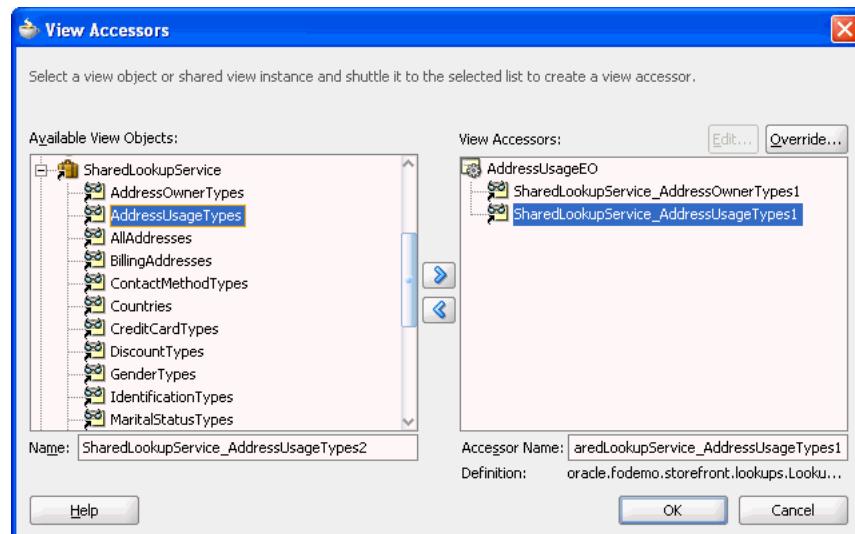
For example, the View Accessors dialog in the Fusion Order Demo application shows the shared application module `LookupServiceAM` with the list of view instances, as shown in [Figure 10–4](#).

The dialog will display all view objects and view instances from your application. If you have not yet enabled application module sharing, you must do so before selecting the view instance. For details, see [Section 10.2.1, "How to Create a Shared Application Module Instance"](#).

By default, the view accessor you create will display the same name as the view object instance (or will have an integer appended when it is necessary to distinguish it from a child view object of the same name). You can edit **Accessor Name** to give it a unique name.

For example, the View Accessors dialog in [Figure 10–4](#) shows the view accessor `Shared_AddressOwnerTypesVA` for the `AddressOwnerTypes` view instance selection in the shared application module `LookupServiceAM`. This view accessor is created on the base entity object `AddressUsagesEO` and accesses the row set of the `AddressOwnerTypes` view instance.

Figure 10–4 Defining a View Accessor on an Entity Object



4. Optionally, if you want to apply an existing view criteria to filter the accessor, with the view accessor selected in the overview editor, click the **Edit** icon.

In the Edit View Accessor dialog, click **Edit** and perform the following steps to apply the view criteria:

- a. Select the view criteria that you want to apply and shuttle it to the **Selected** list.
You can add additional view criteria to apply multiple filters (a logical AND operation will be performed at runtime).
- b. Enter the attribute name for the bind variable that defines the controlling attribute for the view accessor row set.

Unlike view criteria that you set directly on a view object (to create a view instance, for example), the controlling attribute of the view accessor's view criteria derives the value from the view accessor's base view object.

- c. Click **OK** to return to the View Accessors dialog.
5. Click **OK**.

10.4.2 How to Validate Against a View Accessor

View accessors that you create to access the view rows of a destination view object may be used to verify data that your application solicits from the end user at runtime. For example, when the end user fills out a registration form, individual validation rules can verify the title, marital status, and contact code against lookup table data queried by view instances of the shared application module.

You can apply view accessors you have defined on the entity object to these built-in declarative validation rules:

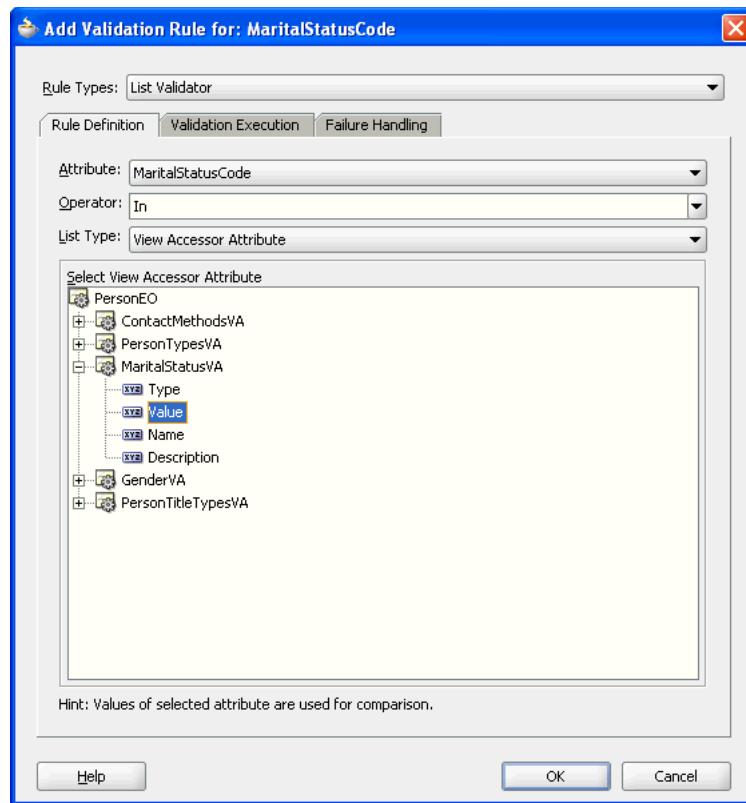
- The Compare validator performs a logical comparison between an entity attribute and a value. When you specify a view accessor to determine the possible values, the compare validator applies the Equals, NotEquals, GreaterThan, LessThan, LessOrEqualTo, GreaterOrEqualTo operator you select to compare against the values returned by the view accessor.
- The List validator compares an entity attribute against a list of values. When you specify a view accessor to determine the valid list values, the List validator applies an In or NotIn operator you select against the values returned by the view accessor.
- The Collection validator performs a logical comparison between an operation performed on a collection attribute and a value. When you specify a view accessor to determine the possible values, the Collection validator applies the Sum, Average, Count, Min, Max operation on the selected collection attribute to compare against the values returned by the view accessor.

Validation rules that you define to allow runtime validation of data for entity-based view objects are always defined on the attributes of the entity object. You use the editor for the entity object to define the validation rule on individual attributes. Any view object that you later define that derives from an entity object with validation rules defined will automatically receive attribute value validation.

To validate against a view accessor comparison, list, or collection type:

1. In the Application Navigator, double-click the desired entity object.
2. In the overview editor, in the **Validation** row, click the **Add** icon.
3. In the Add Validation Rule dialog, in the **Rule Type** dropdown list, select **Compare**, **List**, or **Collection**.
4. Make the selections required by the validator selection.
5. In the **Compare With** or **List Type** dropdown list, select **View Accessor Attribute**.
6. In the **Select View Accessor Attribute** group box, expand the desired view accessor from the shared service and select the attribute you want to provide as validation.

Figure 10–5 shows what the dialog looks like when you use a List validator to select a view accessor attribute.

Figure 10–5 List Validator Using a View Accessor

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

10.4.3 What Happens When You Validate Against a View Accessor

When you use a Compare validator, a `<ListValidationBean>` tag is added to an entity object's XML file. [Example 10–5](#) shows the XML code for the `MaritalStatusCode` attribute in the `Person` entity object. A List validator has been used to validate the user's entry against the list of marital status codes as retrieved by the view accessor from the `LookupsBaseVO` view instance.

Example 10–5 List Validator with View Accessor List Type XML Code

```

<Attribute
    Name="MaritalStatusCode"
    IsUpdateable="true"
    IsNotNull="true"
    Precision="30"
    ColumnName="MARITAL_STATUS_CODE"
    Type="java.lang.String"
    ColumnType="VARCHAR2"
    SQLType="VARCHAR"
    TableName="PERSONS">
    <DesignTime>
        <Attr Name="_DisplaySize" Value="30"/>
    </DesignTime>
    <ListValidationBean
        xmlns="http://xmlns.oracle.com/adfm/validation">

```

```

Name="MaritalStatusCode_Rule_0"
OnAttribute="MaritalStatusCode"
OperandType="VO_USAGE"
Inverse="false"
ViewAccAttrName="Value"
ViewAccName="MaritalStatusVA"/>
</Attribute>

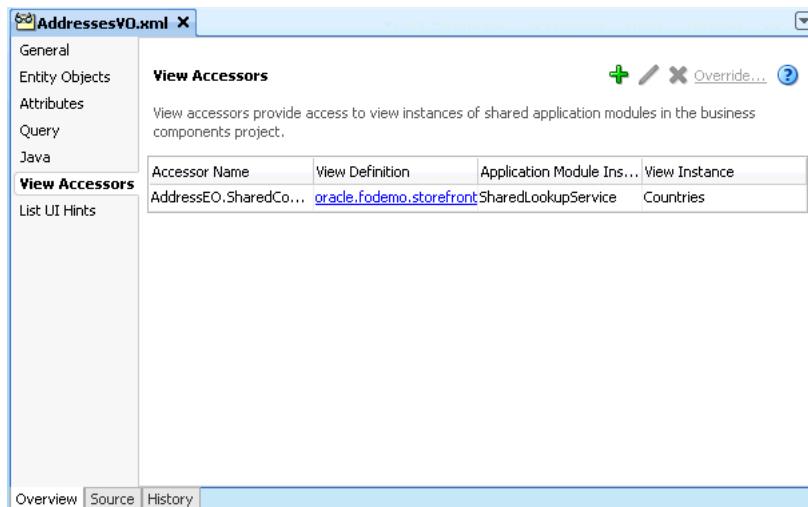
```

10.4.4 How to Create an LOV Based on a Lookup Table

View accessors that you create to access the view rows of a destination view object may be used to display a list of values to the end user at runtime. You first create a view accessor with the desired view instance as its data source, and then you can add the view accessor to an LOV-enabled attribute of the displaying view object. You will edit the view accessor definition for the LOV-enabled attribute so that it points to the specific lookup attribute of the view instance. Because you want to populate the row set cache for the query with static data, you would locate the destination view instance in a shared application module.

While the list usage is defined on the attribute of a view object bound to a UI list control, the view accessor definition exists on either the view object or the view object's base entity object. If you choose to create the view accessor on the view object's entity object, the View Accessors page of the overview editor for the view object will display the inherited view accessor, as shown in [Figure 10–6](#). Alternatively, if you choose to create the view accessor on the attribute's view object, you can accomplish this from either the editor for the LOV definition or from the View Accessors page of the overview editor.

Figure 10–6 View Accessors Page of the Overview Editor



For additional examples of how to work with LOV-enabled attributes, see [Section 5.11, "Working with List of Values \(LOV\) in View Object Attributes"](#).

To create an LOV that displays values from a lookup table:

1. In the Application Navigator, right-click the view object that contains the desired attribute and choose **Open ViewObjectName**.
2. In the overview editor navigation list, select **View Accessors** and check to see whether the view object inherited the desired view accessor from its base entity

object. If no view accessor is present, either create the view accessor on the desired entity object or click the **Create New View Accessors** icon to add the accessor to the view object you are currently editing.

Validation rules that you define are always defined on the attributes of the view object's base entity object. It may therefore be convenient to define view accessors at the level of the base entity objects when you know that you will also validate entity object attributes using a view accessor list.

For details about creating a view accessor, see [Section 10.4.1, "How to Create a View Accessor for an Entity Object or View Object"](#).

3. In the overview editor navigation list, select **Attributes** and select the attribute that is to display the LOV.
4. In the Attributes page, expand the List of Values section, and click the **Add** icon.
5. In the List of Values dialog, select the view accessor from the **List Data Source** dropdown list.

The view accessor you select, will be the one created for the lookup table view object instances to use as the data source.

6. Select the attribute from this view accessor from the **List Attribute** dropdown list that will return the list of values for the attribute you are currently editing.

The editor creates a default mapping between the view object attribute and the LOV-enabled attribute. In this use case, the attributes are the same. For example, the attribute `OrderId` from the `OrdersView` view object would map to the attribute `OrderId` from the `Shared_OrdersVA` view accessor.

7. Optionally, when you want to specify supplemental values that your list returns to the base view object, click **Add** icon in **List Return Values** and map the desired view object attributes to the same attributes accessed by the view accessor. Supplemental attribute return values are useful when you do not require the user to make a list selection for the attributes, yet you want those attributes values, as determined by the current row, to participate in the update.

For example, to map the attribute `StartDate` from the `OrdersView` view object, you would choose the attribute `StartDate` from the `Shared_OrdersVA` view accessor. Do not remove the default attribute mapping for the attribute for which the list is defined.

8. Click **OK**.

10.4.5 What Happens When You Define an LOV for a View Object Attribute

When you add an LOV to a view object attribute, JDeveloper updates the view object's XML file with an `LOVName` property in the `<ViewAttribute>` element. The definition of the LOV appears in a new `<ListBinding>` element. The metadata in [Example 10–6](#) shows that the `MaritalStatusCode` attribute refers to the `MaritalStatusLOV` LOV and sets the choice control type to display the LOV. The LOV definition for `MaritalStatusLOV` appears in the `<ListBinding>` element.

Example 10–6 View Object with LOV List Binding XML Code

```
<ViewObject
    xmlns="http://xmlns.oracle.com/bc4j"
    Name="CustomerRegistrationVO"
    ...
    <ViewAttribute Name="MaritalStatusCode" IsNotNull="true" PrecisionRule="true"
        EntityAttrName="MaritalStatusCode" EntityUsage="PersonEO"
```

```

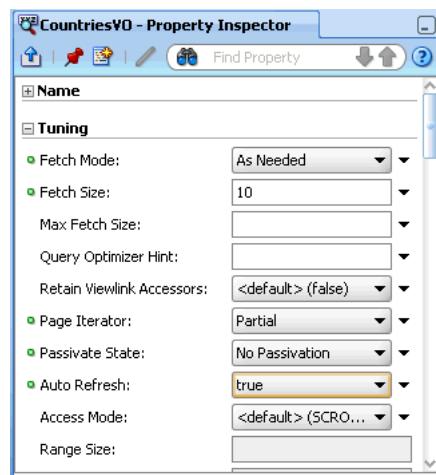
        AliasName="MARITAL_STATUS_CODE"
        LOVName="MaritalStatusCodeLOV">
    <Properties>
        <SchemaBasedProperties>
            <CONTROLYTYPE Value="choice"/>
        </SchemaBasedProperties>
    </Properties>
</ViewAttribute>
...
<ListBinding
    Name="MaritalStatusLOV"
    ListVOName="PersonEO.MaritalStatusVA"
    ListRangeSize="-1"
    NullValueFlag="0"
    NullValueId="LOVUIHints_NullValueId"
    MRUCount="0">
    <AttrArray Name="AttrNames">
        <Item Value="MaritalStatusCode"/>
    </AttrArray>
    <AttrArray Name="ListAttrNames">
        <Item Value="Value"/>
    </AttrArray>
    <AttrArray Name="ListDisplayAttrNames">
        <Item Value="Name"/>
    </AttrArray>
</ListBinding>
...
</ViewObject>

```

10.4.6 How to Automatically Refresh the View Object of the View Accessor

If you need to ensure that your view accessor always queries the latest data from the lookup table, you may be able to set the **Auto Refresh** property on the destination view object, as shown in [Figure 10–7](#). This property allows the view object instance to refresh itself after a change in the database. The typical use case is when you define a view accessor for the destination view object.

Figure 10–7 Property Inspector Show View Object Auto Refresh Option



Note: This feature is currently available only for instances of view objects that you add to a shared application module. In this scenario, the additional overhead to register the notification change listener against the database is warranted.

Because the auto-refresh feature relies on the database change notification feature, observe these restrictions when enabling auto-refresh for your view object:

- The view object must exist in an application module that is a shared application module. This feature will be ignored for all other view object instances.
- The view object must not specify a query with an ORDER BY clause.
- The view object must not query against DATE type data.
- The view objects should query as few read-only tables as possible. This will ensure the best performance and prevent the database invalidation queue from becoming too large.
- The database user must have database notification privileges. For example, to accomplish this with a SQL*Plus command use `grant change notification to <user name>`.

When these restrictions are observed, the refresh is accomplished through the Oracle database change notification feature. Prior to executing the view object query, the framework will use the JDBC API to register the query for database notifications.

When a notification arrives, the row sets of the corresponding view object instance are marked for refresh during the next checkout of the application module. Because the application module waits until the next checkout, the row set currency of the current transaction is maintained and the end user is not hampered by the update.

To enable auto-refresh for a view instance of a shared application module:

1. In the Application Navigator, double-click the view object that you want to receive database change notifications.
2. In the Property Inspector expand the Tuning section, and select **True** for the **Auto Refresh** property.

10.4.7 What Happens at Runtime

The ADF Business Components runtime adds functionality in the attribute setters of the view row and entity object to facilitate the LOV-enabled attribute behavior. In order to display the LOV-enabled attribute values in the user interface, the LOV facility fetches the data source, and finds the relevant row attributes and mapped target attributes.

10.4.8 What You May Need to Know About Lists

There are several things you may need to know about LOVs that you define for attributes of view objects, including how to propagate LOV-enabled attributes from parent view objects to child view objects (by extending an existing view object) and when to use validators instead of an LOV to manage a list of values.

10.4.8.1 Inheritance of AttributeDef Properties from Parent View Object Attributes

When one view object extends another, you can create the LOV-enabled attribute on the base object. Then when you define the child view object in the overview editor, the

LOV definition will be visible on the corresponding view object attribute. This inheritance mechanism allows you to define an LOV-enabled attribute once and apply it later across multiple view objects instances for the same attribute. For details about extending a view object from another view object definition, see [Section 33.9.2, "How To Extend a Component After Creation"](#).

You can also use the overview editor to extend the inherited LOV definition. For example, you may add extra attributes already defined by the base view object's query to display in selection list. Alternatively, you can create a view object instance that uses a custom WHERE clause to query the supplemental attributes not already queried by the base view object. For information about customizing entity-based view objects, see [Section 5.9, "Working with Bind Variables"](#).

10.4.8.2 Using Validators to Validate Attribute Values

If you have created an LOV-enabled attribute for a view object, there is no need to validate the attribute using a List validator. You use an attribute validator only when you do not want the list to display in the user interface but still need to restrict the list of valid values. A List validator may be a simple static list or it may be a list of possible values obtained through a view accessor you define. Alternatively, you might prefer to use a Key Exists validator when the attribute displayed in the UI is one that references a key value (such as a primary, foreign, or alternate key). For information about declarative validation in ADF Business Components, see [Chapter 7, "Defining Validation and Business Rules Declaratively"](#).

10.5 Testing View Object Instances in a Shared Application Module

JDeveloper includes an interactive application module testing tool that you can use to test all aspects of its data model without having to use your application user interface or write a test client program. Running the Business Component Browser can often be the quickest way of exercising the data functionality of your business service during development.

10.5.1 How to Test the Base View Object Using the Business Component Browser

The application module is the transactional component that the Business Component Browser (or UI client) will use to work with application data. The set of view objects used by an application module defines its data model, in other words, the set of data that a client can display and manipulate through a user interface. You can use the Business Component Browser to test that the accessors you defined yield the expected validation result and that they display the correct LOV attribute values.

To create an application module, use the Create Application Module wizard, which is available in the New Gallery. For more information, see [Section 9.2, "Creating and Modifying an Application Module"](#).

To test the view objects you added to an application module, use the Business Component Browser, which is accessible from the Application Navigator.

To test view objects in an application module configuration:

1. In the Application Navigator, expand the project containing the desired application module and view objects.
2. Right-click the application module and choose **Run**.

Alternatively, choose **Debug** when you want to run the application in the Business Component Browser with debugging enabled. For example, when debugging using the Business Component Browser, you can view status message and

exceptions, step in and out of source code, and manage breakpoints. JDeveloper opens the debugger process panel in the Log window and the various debugger windows.

For details about receiving diagnostic messages specific to ADF Business Component debugging, see [Section 6.3.7, "How to Enable ADF Business Components Debug Diagnostics"](#).

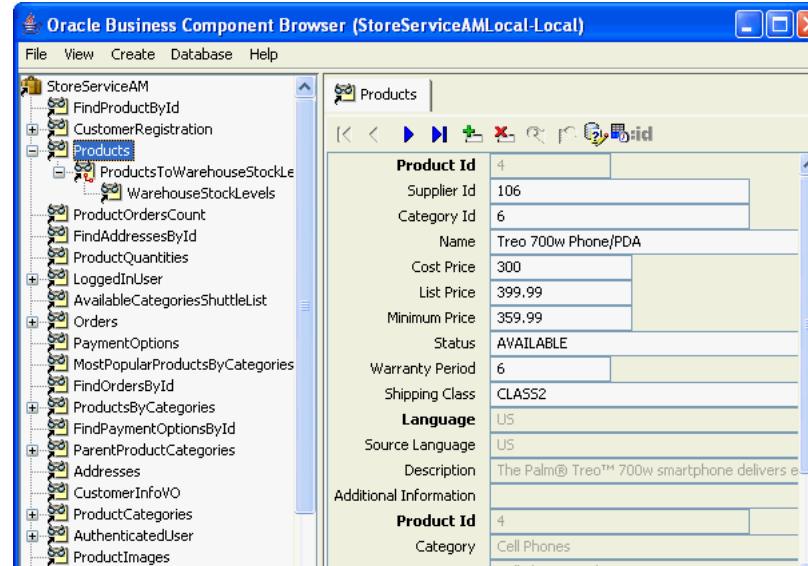
3. In the Select Business Components Configuration dialog, choose the desired application module configuration from the **Business Component Configuration Name** list to run the Business Component Browser.

By default, an application module has only its default configurations, named *AppModuleLocal* and *AppModuleShared*. If you have created additional configurations for your application module and want to test it using one of those instead, just select the desired configuration from the **Business Components Configuration** dropdown list on the Connect dialog before clicking **Connect**.

4. Click **Connect** to start the application module using the selected configuration.
5. To execute a view object in the Business Component Browser, expand the tree list and double-click the desired view object node.

Note that the view object instance may already appear executed in the testing session. In this case, the tester panel on the right already displays query results for the view object instance, as shown in [Figure 10–8](#). The fields in the tester panel of a read-only view object will always appear disabled since the data it represents is not editable.

Figure 10–8 Testing the Data Model in the Business Component Browser



10.5.2 How to Test LOV-Enabled Attributes Using the Business Component Browser

To test the LOV you created for a view object attribute, use the Business Component Browser, which is accessible from the Application Navigator. For details about displaying the Browser and the supported control types, see [Section 5.11.4, "How to Test LOV-Enabled Attributes Using the Business Component Browser"](#).

10.5.3 What Happens When You Use the Business Component Browser

When you launch the Business Component Browser, JDeveloper starts the tester tool in a separate process and the Business Component Browser appears. The tree at the left of the dialog displays all of the view object instances in your application module's data model. [Figure 10–8](#) shows just one instance in the expanded tree, called ProductImages. After you double-click the desired view object instance, the Business Component Browser will display a panel to inspect the query results, as shown in [Figure 10–8](#).

The test panel will appear disabled for any read-only view objects you display because the data is not editable. But even for the read-only view objects, the tool affords some useful features:

- You can validate that the UI hints based on the Label Text control hint and format masks are defined correctly.
- You can also scroll through the data using the toolbar buttons.

The Business Component Browser becomes even more useful when you create entity-based view objects that allow you to simulate inserting, updating, and deleting rows, as described in [Section 6.3.2, "How to Test Entity-Based View Objects Interactively"](#).

10.5.4 What Happens at Runtime When Another Service Accesses the Shared Application Module Cache

When a shared application module with application scope is requested by an LOV, then the ADF Business Components runtime will create an `ApplicationPool` object for that usage. There is only one `ApplicationPool` created for each shared usage that has been defined in the Business Components project file (`.jpx`). The runtime will then use that `ApplicationPool` to acquire an application module instance that will be used like a user application module instance, to acquire data. The reference to the shared application module instance will be released once the application-scoped application module is reset. The module reference is released whenever you perform an unmanaged release or upon session timeout.

Since multiple threads will be accessing the data caches of the shared application module, it is necessary to partition the iterator space to prevent race conditions between the iterators of different sessions. This will help ensure that the next request from one session does not change the state of the iterator that is being used by another session. The runtime uses ADF Business Components support for multiple iterators on top of a single `RowSet` to prevent these race conditions. So, the runtime will instantiate as many iterators as there are active sessions for each `RowSet`.

An application-scoped shared application module lifecycle is similar to the lifecycle of any application module that is managed by the `ApplicationPool` object. For example, once all active sessions have released their shared application module, then the application module may be garbage-collected by the `ApplicationPool` object. The shared pool may be tuned to meet specific application requirements.

Session-scoped shared application modules are simply created as nested application module instances within the data model of the root, user application module. For details about nested application modules, [Section 9.4, "Defining Nested Application Modules"](#).

Using Oracle ADF Model in a Fusion Web Application

This chapter describes how an application module's data model and business service interface methods appear at design time for drag and drop data binding, how they are accessible at runtime by the ADF Model data binding layer using the application module data control, and how developers can use the Data Controls panel to create databound pages.

This chapter includes the following sections:

- [Section 11.1, "Introduction to ADF Data Binding"](#)
- [Section 11.2, "Exposing Application Modules with Oracle ADF Data Controls"](#)
- [Section 11.3, "Using the Data Controls Panel"](#)
- [Section 11.4, "Working with the DataBindings.cpx File"](#)
- [Section 11.5, "Configuring the ADF Binding Filter"](#)
- [Section 11.6, "Working with Page Definition Files"](#)
- [Section 11.7, "Creating ADF Data Binding EL Expressions"](#)
- [Section 11.8, "Using Simple UI First Development"](#)

11.1 Introduction to ADF Data Binding

Oracle ADF Model implements the two concepts in JSR-227 that enable decoupling the user interface technology from the business service implementation: *data controls* and *declarative bindings*.

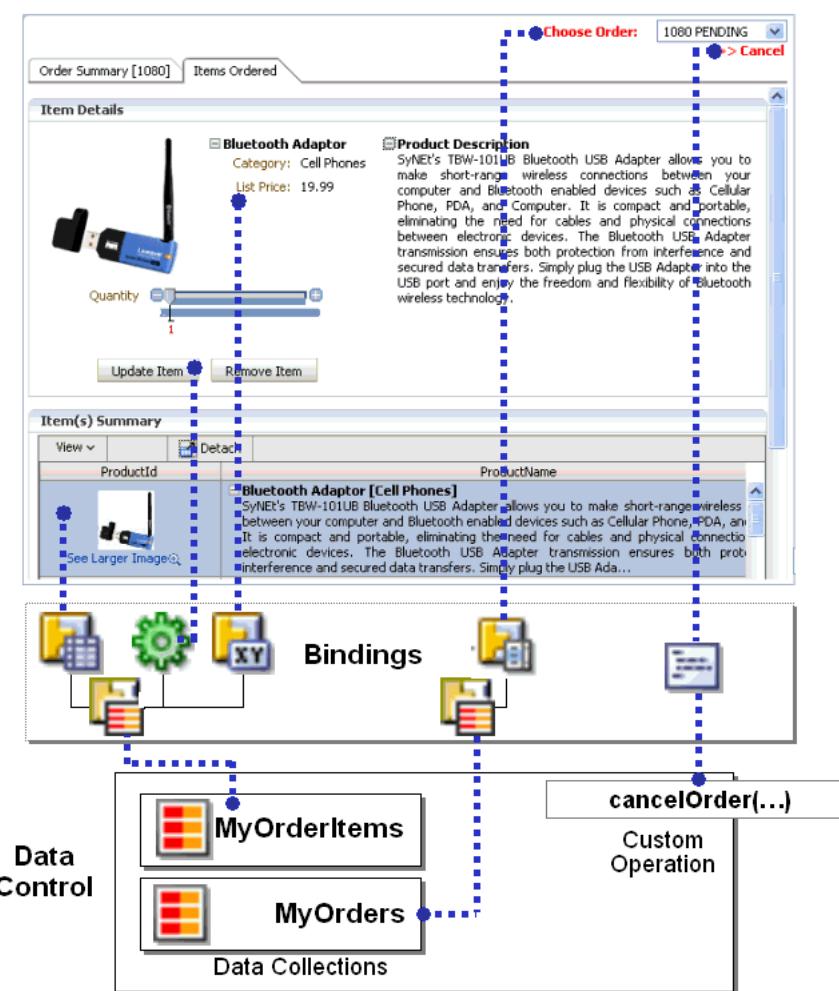
Data controls abstract the implementation technology of a business service by using standard metadata interfaces to describe the service's operations and data collections, including information about the properties, methods, and types involved. In an application that uses business components, a data control is automatically created when you create an application module, and it contains all the functionality of the application module. Developers can then use the representation of the data control displayed in JDeveloper's Data Controls panel to create UI components that are automatically bound to the application module. At runtime, the ADF Model layer reads the information describing the data controls and bindings from appropriate XML files and implements the two-way connection between the user interface and the business service.

Declarative bindings abstract the details of accessing data from data collections in a data control and of invoking its operations. There are three basic kinds of declarative binding objects:

- **Iterator bindings:** Simplify building user interfaces that allow scrolling and paging through collections of data and drilling-down from summary to detail information.
 - **Value bindings:** Used by UI components that display data. Value bindings range from the most basic variety that work with a simple text field to more sophisticated list and tree bindings that support the additional needs of list, table, and tree UI controls.
 - **Action bindings:** Used by UI components like hyperlinks or buttons to invoke built-in or custom operations on data collections or a data control without writing code.

[Figure 11-1](#) shows how bindings connect UI components to data control collections and methods.

Figure 11-1 Bindings Connect UI Components to Data Controls



The group of bindings supporting 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 runtime to instantiate the page's bindings. These bindings are held in a request-scoped map called the *binding container*, accessible during each page request using the EL expression `# {bindings}`. This expression always evaluates to the binding container for the current page.

You can design a databound user interface by dragging an item from the Data Controls panel and dropping it on a page as a specific UI component. When you use data controls to create a UI component, JDeveloper automatically creates the various code and objects needed to bind the component to the data control you selected.

Note: Using the ADF Model layer to perform business service access ensures that the view and the business service stay in sync. For example, while you could call a method on an application module by class casting the data control reference to the application module instance and then calling the method directly, doing so will bypass the model layer and it will become unaware of any changes produced by this approach.

11.2 Exposing Application Modules with Oracle ADF Data Controls

The application module data control is a thin adapter over an application module pool that automatically acquires an available application module instance at the beginning of the request. During the current request, the application module data control holds a reference to the application module instance on behalf of the current user session. At the end of the request, the data control releases the instance back to the pool. Importantly, the application module component directly implements the interfaces that the binding objects expect for data collections, built-in operations, and service methods. This allows the bindings to work *directly* with the application module instances in its data model. Specifically, this optimized interaction allows:

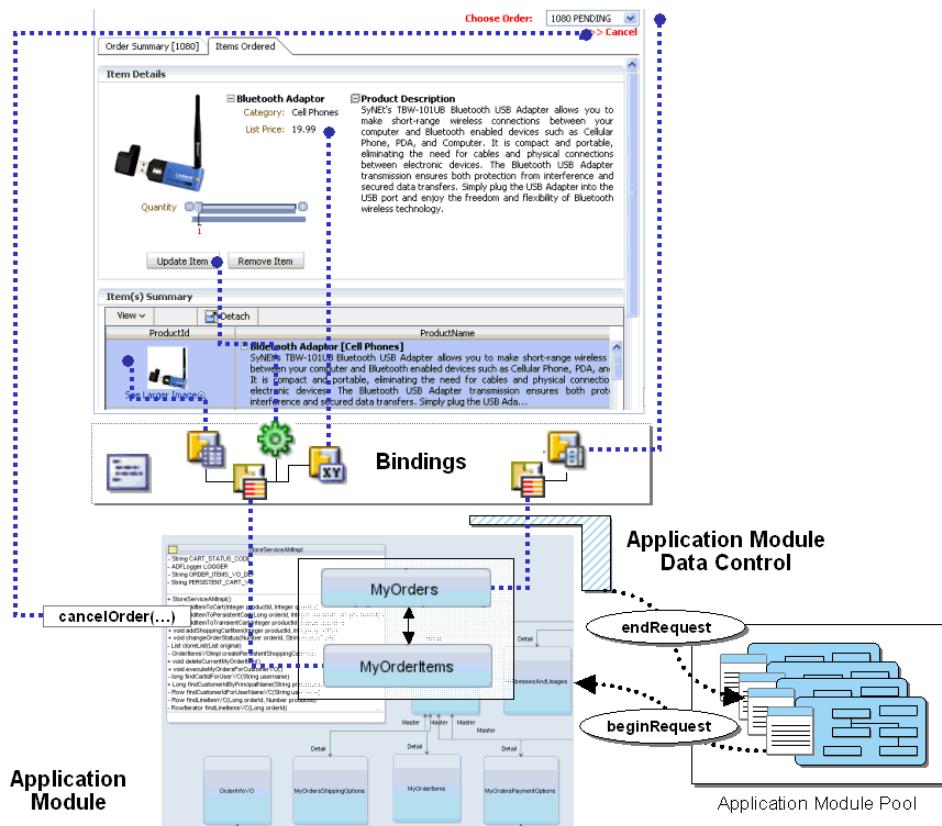
- Iterator bindings to directly bind to the default row set iterator of the default row set of any view object instance. The row set iterator manages the current object and current range information.

Tip: You can also use the iterator binding to bind to a secondary named row set that you have created. To bind to a secondary row set, you need to use the `RSName` attribute on the binding. For more information about the difference between the default row set and secondary row sets and how to create them, see [Section 35.1.9, "Working with Multiple Row Sets and Row Set Iterators"](#).

- Action bindings to directly bind to either:
 - Custom methods on the data control client interface
 - Built-in operations of the application module and view objects

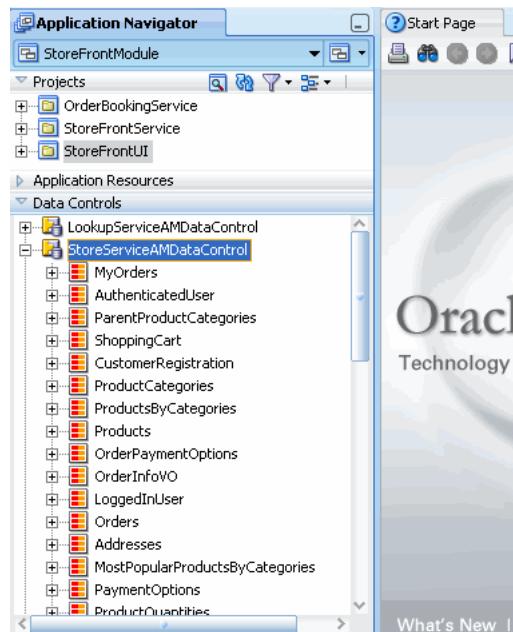
[Figure 11-2](#) illustrates the pool management role the application module data control plays and highlights the direct link between the bindings and the application module instance.

Figure 11–2 Bindings Connect Directly to View Objects and Methods of an Application Module from a Pool



11.2.1 How an Application Module Data Control Appears in the Data Controls Panel

You use the Data Controls panel to create databound HTML elements (for JSP pages), and databound UI components (for JSF JSP pages) by dragging and dropping icons from the panel onto the visual editor for a page. [Figure 11–3](#) shows the Data Controls panel displaying the data controls for the StoreFront module.

Figure 11–3 Data Controls Panel in JDeveloper

The Data Controls panel lists all the data controls that have been created for the application's business services and exposes all the collections (row set of data objects), methods, and built-in operations that are available for binding to UI components.

Note: If you've configured JDeveloper to expose them, any view link accessor returns are also displayed. For more information, see [Section 5.6, "Working with Multiple Tables in a Master-Detail Hierarchy"](#). To view the accessor methods:

1. In the JDeveloper main menu, choose **Tools > Preferences**.
 2. Select the **Data Controls Panel** node.
 3. Select **Show Underlying Accessor Nodes** to activate the checkbox.
-

For example, in an application that uses ADF Business Components to define the business services, each data control on the Data Controls panel represents a specific application module, and exposes the view object instances in that application's data model. The hierarchy of objects in the data control is defined by the view links between view objects that have specifically been added to the application module data model. For information about creating view objects and view links, see [Chapter 5, "Defining SQL Queries Using View Objects"](#). For information about adding view links to the data model, see [Section 5.6.4, "How to Enable Active Master-Detail Coordination in the Data Model"](#).

Tip: You can open the overview editor for a view object by right-clicking the associated data control object and choosing **Edit Definition**.

For example, the `StoreServiceAMDataControl` application module implements the business service layer of the `StoreFront` module application. Its data model contains numerous view object instances, including several master-detail hierarchies. The view layer of the ADF sample application consists of JSF pages whose UI components are bound to data from the view object instances in the

StoreServiceAMDataControl's data model, and to built-in operations and service methods on its client interface.

11.2.1.1 How the Data Model and Service Methods Appear in the Data Controls Panel

[Figure 11–4](#) illustrates how the Data Controls panel displays the view object instances in the StoreServiceAMDataControl's data model. Each view object instance appears as a named data collection whose name matches the view object instance name (note that for viewing simplicity, the figure omits some details in the tree that appear for each view object). The Data Controls panel reflects the master-detail hierarchies in your application module data model by displaying detail data collections nested under their master data collection.

The Data Controls panel also displays each custom method on the application module's client interface as a named data control custom operation whose name matches the method name. If a method accepts arguments, they appear in a Parameters node as operation parameters nested inside the operation node.

Figure 11–4 How the Data Model Appears in the Data Controls Panel



11.2.1.2 How Transaction Control Operations Appear in the Data Controls Panel

The application module data control exposes two data control built-in operations named `Commit` and `Rollback`, as shown in [Figure 11–5](#) (note that the `Operations` node in the data controls tree omits all of the data collections and custom operations for a more streamlined view). At runtime, when these operations are invoked by the data binding layer, they delegate to the `commit()` and `rollback()` methods of the `Transaction` object associated with the current application module instance.

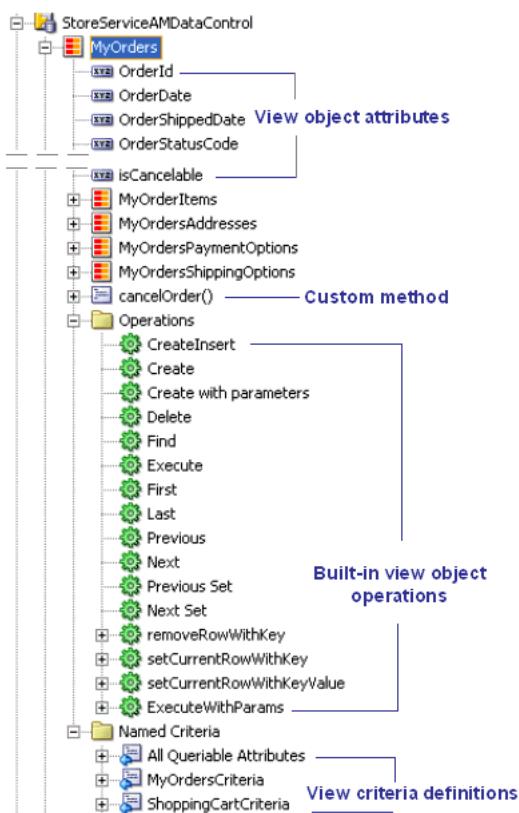
Note: In an application module with many view object instances and custom methods, you may need to scroll the Data Controls panel display to find the `Operations` node that is the direct child node of the data control. This node is the one that contains these built-in operations.

Figure 11–5 How Transaction Control Operations Appear in the Data Controls panel

11.2.1.3 How View Objects Appear in the Data Controls Panel

[Figure 11–6](#) shows how each view object instance in the application module's data model appears in the Data Controls panel. The view object attributes are displayed as immediate child nodes of the corresponding data collection, as are any custom methods you've created. If you have selected any custom methods to appear on the view object's client interface, they appear as custom methods immediately following the view object attributes at the same level. If the method accepts arguments, these appear in a nested Parameters node as operation parameters.

By default, implicit view criteria are created for each attribute that is able to be queried on a view object, and appear as the All Queriable Attributes are displayed under the Named Criteria node, as shown in [Figure 11–6](#). If any named view criteria were created for the view object, they appear under the Named Criteria node. The View Criteria expressions (both implicit and named) appear as method returns. The conjunction used in the query, along with the criteria items and if applicable, any nested criteria, are shown as children. These items are used to create quick search forms, as detailed in [Chapter 25, "Creating ADF Databound Search Forms"](#).

Figure 11–6 How View Objects Appear in the Data Controls Panel

As shown in [Figure 11–6](#), the Operations node under the data collection displays all its available built-in operations. If an operation accepts one or more parameters, then those parameters appear in a nested Parameters node. At runtime, when one of these data collection operations is invoked by name by the data binding layer, the application module data control delegates the call to an appropriate method on the `ViewObject` interface to handle the built-in functionality. The built-in operations fall into three categories: operations that affect the current row, operations that refresh the data collection, and all other operations.

Operations that affect the current row:

- `CreateInsert`: Creates a new row that becomes the current row, and inserts the new blank row into the data source.
- `Create`: Creates a new row that becomes the current row, but does not insert it.
- `Create with Parameters`: Creates a new row taking parameter values. The passed parameters can supply the create-time value of the discriminator or composing parent's foreign key attributes that are required at create time for polymorphic view object and for a composed child view object row when not created in the context of a current view linked parent row, respectively. For more information about polymorphic view objects, see [Section 35.7, "Using View Objects to Work with Multiple Row Types"](#).
- `Delete`: Deletes the current row.
- `First`: Sets the current row to be the first row in the row set.
- `Last`: Sets the current row to be the last row in the row set.
- `Previous`: Sets the current row to be the previous row in the row set.
- `Next`: Sets the row to be the next row in the row set.
- `Previous Set`: Navigates forward one full set of rows.
- `Next Set`: Navigates backward one full set of rows.
- `setCurrentRowWithKey`: Tries to finds a row using the serialized string representation of row key passed as a parameter. If found, that row becomes the current row.
- `setCurrentRowWithValue`: Tries to finds a row using the primary key attribute value passed as a parameter. If found, that row becomes the current row.

Operations that refresh the data collection:

- `Execute`: Refreshes the data collection by executing or reexecuting the view object's query, leaving any bind parameters at their current values.
- `ExecuteWithParams`: Refreshes the data collection by first assigning new values to the named bind variables passed as parameters, then executing or reexecuting the view object's query.

Note: The `executeWithParams` operation appears only for view objects that have defined one or more named bind variables at design time.

All other operations:

- `removeRowWithKey`: Tries to finds a row using the serialized string representation of row key passed as a parameter. If found, the row is removed.

- Find: Toggles "Find Mode" on and off for the data collection.

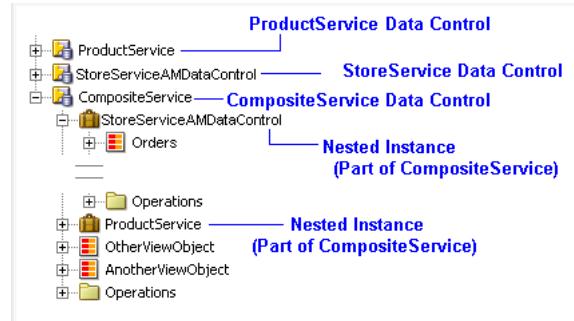
11.2.1.4 How Nested Application Modules Appear in the Data Controls Panel

If you build composite application modules by including nested instances of other application modules, the Data Controls panel reflects this component assembly in the tree hierarchy. For example, assume that in addition to the `StoreServiceAMDataControl` application module that you have also created the following application modules in the same package:

- An application module named `ProductService`, and renamed its data control to `ProductService`
- An application module named `CompositeService`, and renamed its data control to `CompositeService`

Then assume that you've added two view object instances named `OtherViewObject` and `AnotherViewObject` to the data model of `CompositeService` and that on the Application Modules page of the Edit Application Module dialog you have added an instance of the `StoreServiceAMDataControl` application module and an instance of the `ProductService` application module to reuse them as part of `CompositeService`. [Figure 11–7](#) illustrates how your `CompositeService` would appear in the Data Controls panel (note that much of the structure of the nested `StoreServiceAMDataControl` has been omitted for clarity). The nested instances of `StoreServiceAMDataControl` and `ProductService` appear in the panel tree display nested inside of the `CompositeService` data control. The entire data model and set of client methods that the nested application module instances expose to clients are automatically available as part of the `CompositeService` that reuses them.

Figure 11–7 How Nested Application Modules Appear in the Data Controls Panel



One possibly confusing point is that even though you have reused nested instances of `StoreServiceAMDataControl` and `ProductService` inside of `CompositeService`, the `StoreServiceAMDataControl` and `ProductService` application modules also appear themselves as top-level data control nodes in the panel tree. JDeveloper assumes that you might want to sometimes use `StoreServiceAMDataControl` or `ProductService` on their own as separate data controls from `CompositeService`, so it displays all three of them. You need to be careful to perform your drag-and-drop data binding from the correct data control. If you want your page to use a view object instance from the nested `StoreServiceAMDataControl` instance's data model that is an aggregated part of the `CompositeService` data control, then ensure that you select the data collection that appears as part of the `CompositeService` data control node in the panel.

It is important to do the drag-and-drop operation that corresponds to your intended usage. When you drop a data collection from the *top-level*

StoreServiceAMDataControl data control node in the panel, at runtime your page will use an instance of the StoreServiceAMDataControl application module acquired from a pool of StoreServiceAMDataControl components. When you drop a data collection from the *nested* instance of StoreServiceAMDataControl that is part of CompositeService, at runtime your page will use an instance of the CompositeService application module acquired from a pool of CompositeService components. Since different types of application module data controls will have distinct transactions and database connections, inadvertently mixing and matching data collections from both a nested application module and a top-level data control will lead to unexpected runtime behavior.

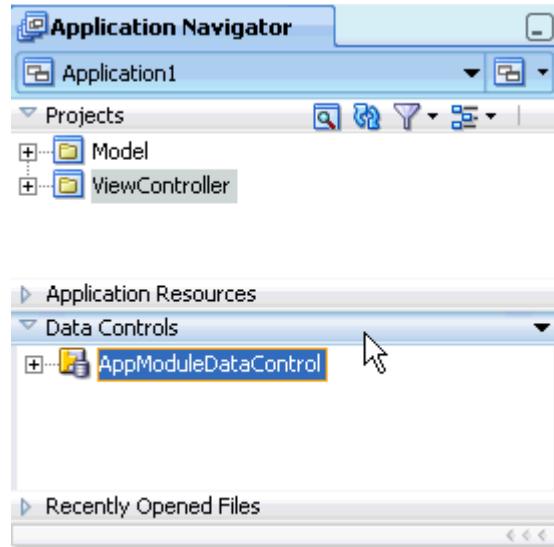
11.2.2 How to Open the Data Controls Panel

The Data Controls Panel is a panel within the Application Navigator, located at the top left of JDeveloper. To view the contents, click the panel header to expand the panel. If you do not see the panel header, then the Application Navigator may not be displaying.

To open the Application Navigator and Data Controls panel:

1. From the JDeveloper main menu, choose **View > Application Navigator**.
2. To open the Data Controls accordion panel, click the expand icon in the Data Controls header, as shown in [Figure 11–8](#).

Figure 11–8 Data Controls Panel in the Application Navigator



11.2.3 How to Refresh the Data Control

Any time changes are made to the application module or underlying services, you need to manually refresh the data control in order to view the changes.

To refresh the application module data control:

1. Right-click the application module data control.
2. From the context menu, choose **Refresh**.

When you click **Refresh**, the Data Controls panel looks for all available data controls, and therefore will now reflect any structural changes made to the data control.

11.2.4 Packaging a Data Control for Use in Another Project

You can package up data controls so that they can be used in another project. For example, one development group might be tasked with creating the services and data controls, while another development group might be tasked with creating the UI. The first group would create the services and data controls, and then package them up as an Oracle ADF Library and send it to the second group. The second group can then add the data controls to their project using the JDeveloper Resource Palette. For more information, see [Chapter 31, "Reusing Application Components"](#).

11.3 Using the Data Controls Panel

You can design a databound user interface by dragging an item from the Data Controls panel and dropping it on a page as a specific UI component. When you use data controls to create a UI component, JDeveloper automatically creates the various code and objects needed to bind the component to the data control you selected.

In the Data Controls panel, each data control object is represented by a specific icon. [Table 11–1](#) describes what each icon represents, where it appears in the Data Controls panel hierarchy, and what components it can be used to create.

Table 11–1 Data Controls Panel Icons and Object Hierarchy

Icon	Name	Description	Used to Create...
	Data Control	<p>Represents a data control. You cannot use the data control itself to create UI components, but you can use any of the child objects listed under it. Depending on how your business services were defined, there may be more than one data control.</p> <p>Usually, there is one data control for each application module. However, you may have additional data controls that were created for other types of business services (for example, for web services). For information about creating data controls for web services, see Chapter 12, "Integrating Web Services Into a Fusion Web Application".</p>	Serves as a container for the other object and is not used to create anything.
	Collection	<p>Represents a named data collection. A <i>data collection</i> represents a set of data objects (also known as a <i>row set</i>) in the data model. Each object in a data collection represents a specific structured data item (also known as a <i>row</i>) in the data model. Throughout this guide, <i>data collection</i> and <i>collection</i> are used interchangeably.</p> <p>For application modules, the data collection is the default row set contained in a view object instance. The name of the collection matches the view object instance name.</p> <p>A view link creates a master-detail relationship between two view objects. If you explicitly add an instance of a detail view object (resulting from a view link) to the application module data model, the collection contained in that detail view object appears as a child of the collection contained in the master view object. For information about adding detail view objects to the data model, see Section 5.6.4, "How to Enable Active Master-Detail Coordination in the Data Model".</p> <p>The children under a collection may be attributes of the collection, other collections that are related by a view link, custom methods that return a value from the collection, or built-in operations that can be performed on the collection.</p> <p>If you've configured JDeveloper to display viewlink accessor returns, then those are displayed as well.</p>	<p>Forms, tables, graphs, trees, range navigation components, and master-detail components.</p> <p>For more information about using collections on a data control to create forms, see Chapter 20, "Creating a Basic Databound Page".</p> <p>For more information about using collections to create tables, see Chapter 21, "Creating ADF Databound Tables".</p> <p>For more information about using master-detail relationships to create UI components, see Chapter 22, "Displaying Master-Detail Data".</p> <p>For information about creating graphs, charts, and other visualization UI components, see Chapter 24, "Creating Databound ADF Data Visualization Components".</p>
	Attribute	<p>Represents a discrete data element in an object (for example, an attribute in a row). Attributes appear as children under the collections or method returns to which they belong.</p> <p>Only the attributes that were included in the view object are shown under a collection. If a view object joins one or more entity objects, that view object's collection will contain selected attributes from all of the underlying entity objects.</p>	<p>Label, text field, date, list of values, and selection list components.</p> <p>For information about using attributes to create fields on a page, see Section 20.2, "Using Attributes to Create Text Fields".</p> <p>For information about creating lists, see Chapter 23, "Creating Databound Selection Lists and Shuttles".</p>

Table 11–1 (Cont.) Data Controls Panel Icons and Object Hierarchy

Icon	Name	Description	Used to Create...
	Structured Attribute	<p>Represents a returned object that is neither a Java primitive type (represented as an attribute) nor a collection of any type. An example of a structured attribute would be a domain, which is a developer-created data type used to simplify application maintenance.</p> <p>For more information about domains, see Section 34.1, "Creating Custom, Validated Data Types Using Domains".</p>	Label, text field, date, list of values, and selection list components.
	Method	<p>Represents an operation in the data control or one of its exposed structures that may accept parameters, perform some business logic and optionally return single value, a structure, or a collection.</p> <p>In application module data controls, custom methods are defined in the application module itself and usually return either nothing or a single scalar value. For more information about creating custom methods, see Chapter 9, "Implementing Business Services with Application Modules".</p>	Command components For methods that accept parameters: command components and parameterized forms. For more information about using methods that accept parameters, see Section 26.2.2.1, "Using Parameters in a Method" .
	Method Return	<p>Represents an object that is returned by a custom method. The returned object can be a single value or a collection.</p> <p>If a custom method defined in the application module returns anything at all, it is usually a single scalar value. Application module methods do not need to return a set of data to the view layer, because displaying the latest changes to the data is handled by the view objects in the data model (for more information, see Section 3.4, "Overview of the UI-Aware Data Model"). However, custom methods in non-application module data controls (for example, a data control for a CSV file) can return collections to the view layer.</p> <p>A method return appears as a child under the method that returns it. The objects that appear as children under a method return can be attributes of the collection, other methods that perform actions related to the parent collection, or operations that can be performed on the parent collection.</p>	The same components as for collections and attributes. For named criteria: query or quick query forms. For more information, see Chapter 25, "Creating ADF Databound Search Forms" . When a single-value method return is dropped, the method is not invoked automatically by the framework. You need either to create an invoke action as an executable, or to drop the corresponding method as a button to invoke the method. For more information about executables, see Section 11.6.2.2, "Executable Binding Objects" .
	Operation	<p>Represents a built-in data control operation that performs actions on the parent object. Data control operations are located in an Operations node under collections or method returns, and also under the root data control node. The operations that are children of a particular collection or method return operate on those objects only, while operations under the data control node operate on all the objects in the data control.</p> <p>If an operation requires one or more parameters, they are listed in a Parameters node under the operation.</p>	UI command components, such as buttons, links, and menus. For more information, see Section 20.4, "Incorporating Range Navigation into Forms" and Section 20.5, "Creating a Form to Edit an Existing Record" .
	Parameter	Represents a parameter value that is declared by the method or operation under which it appears. Parameters appear in the Parameters node under a method or operation.	Label, text, and selection list components.

11.3.1 How to Use the Data Controls Panel

JDeveloper provides you with a predefined set of UI components from which to choose for each data control item you can drop.

To use the Data Controls panel to create UI components:

1. Select an item in the Data Controls panel and drag it onto the visual editor for your page. For a definition of each item in the panel, see [Table 11–1](#).

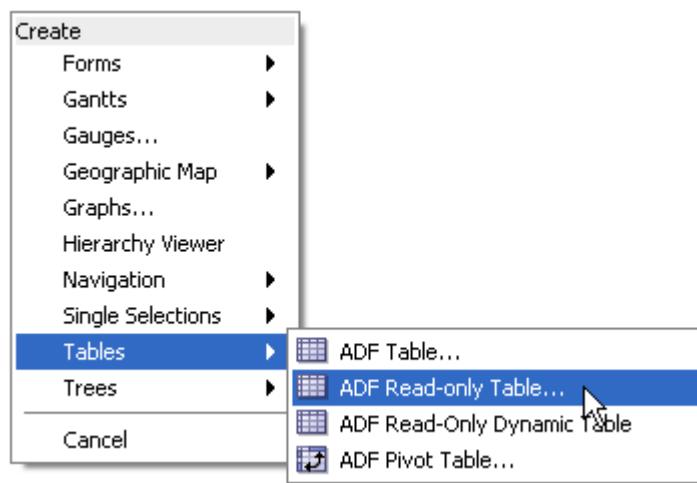
Tip: If you need to drop an operation or method onto a method activity in a task flow, you can simply drag and drop it onto the activity in the diagram.

2. From the ensuing context menu, select a UI component.

When you drag an item from the Data Controls panel and drop it on a page, JDeveloper displays a context menu of all the default UI components available for the item you dropped. The components displayed are based on the libraries in your project.

[Figure 11–9](#) shows the context menu displayed when a data collection from the Data Controls panel is dropped on a page.

Figure 11–9 Data Controls Panel Context Menu



Depending on the component you select from the context menu, JDeveloper may display a dialog that enables you to define how you want the component to look. For example, if you select **ADF Read-only Table** from the context menu, the Edit Table Columns dialog launches. This dialog enables you to define which attributes you want to display in the table columns, what the column labels are, what types of text fields you want use for each column, and what functionality you want to include, such as row selection or column sorting. For more information about creating tables, see [Chapter 21, "Creating ADF Databound Tables"](#).

The UI components selected by default are determined first by any UI control hints set on the corresponding business object. If no control hints have been set, then JDeveloper uses input components for standard forms and tables, and output components for read-only forms and tables. Components for lists are determined based on the type of list you chose when dropping the data control object.

Once you select a component, JDeveloper inserts the UI component on the page in the visual editor. For example, if you drag a collection from the Data Controls panel and choose **ADF Read-only Table** from the context menu, a read-only table appears in the visual editor, as shown in [Figure 11–10](#).

Figure 11–10 Databound UI Component: ADF Read-Only Table

# {bindings.Products.hints.ProductId.label}	# {bindings.Products.hints.SupplierId.label}	# {bindings.Products.hints.CategoryId.label}	# {bindings.Products.hints.ProductName.label}	# {ProdCost!}
#{row.ProductId}	#{row.SupplierId}	#{row.CategoryId}	#{row.ProductName}	#{row.Cost!}
#{row.ProductId}	#{row.SupplierId}	#{row.CategoryId}	#{row.ProductName}	#{row.Cost!}
#{row.ProductId}	#{row.SupplierId}	#{row.CategoryId}	#{row.ProductName}	#{row.Cost!}

By default, the UI components created when you use the Data Controls panel use ADF Faces Rich Client components, are bound to attributes in the ADF data control, and may have one or more built-in features, including:

- Databound labels
- Tooltips
- Formatting
- Basic navigation buttons
- Validation, if validation rules are attached to a particular attribute. For more information, see [Chapter 7, "Defining Validation and Business Rules Declaratively"](#).

The default components are fully functional without any further modifications. However, you can modify them to suit your particular needs. Each component and its various features are discussed further in [Part IV, "Creating a Databound Web User Interface"](#).

Tip: If you want to change the type of ADF databound component used on a page, the easiest method is to delete the component and drag and drop a new one from the Data Controls panel. When you delete a databound component from a page, if the related binding objects in the page definition file are not referenced by any other component, JDeveloper automatically deletes those binding objects for you.

11.3.2 What Happens When You Use the Data Controls Panel

When an Oracle ADF web application is built using the JSF framework, it requires a few additional application object definitions to render and process a page containing ADF databound UI components. If you do not use the Data Controls panel, you will have to manually configure these various files yourself. However, when you use the Data Controls panel, JDeveloper does all of the following required steps:

- Creates a `DataBindings.cpx` file in the default package for the project (if one does not already exist), and adds an entry for the page.

`DataBindings.cpx` files define the *binding context* for the application. The binding context is a container object that holds a list of available data controls and data binding objects. For more information, see [Section 11.3.3, "What Happens at Runtime: How the Binding Context Works"](#). Each `DataBindings.cpx` file maps individual pages to the binding definitions in the page definition file and registers

the data controls used by those pages. For more information, see [Section 11.4, "Working with the DataBindings.cpx File"](#).

- Creates the `adf.m.xml` file in the `META-INF` directory. This file creates a registry for the `DataBindings.cpx` file, which allows the application to locate it at runtime so that the binding context can be created.
- Registers the ADF binding filter in the `web.xml` file.

The ADF binding filter preprocesses any HTTP requests that may require access to the binding context. For more information about the binding filter configuration, see [Section 11.5, "Configuring the ADF Binding Filter"](#).
- Creates the `orion-application.xml` file and adds a reference to the Oracle ADF shared libraries needed by the application
- Adds the following libraries to the view project:
 - ADF Faces Databinding Runtime
 - Oracle XML Parser v2
 - JDeveloper Runtime
 - SQLJ Runtime
 - ADF Model Runtime
 - BC4J Runtime
 - Oracle JDBC
 - Connection Manager
 - BC4J Oracle Domains
- Adds a page definition file (if one does not already exist for the page) to the page definition subpackage, the name of which is defined in the ADFm settings of the project properties. The default subpackage is `view.pageDefs` in the `adfmsrc` directory.

The page definition file (`pageNamePageDef.xml`) defines the ADF binding container for each page in an application's view layer. The binding container provides runtime access to all the ADF binding objects for a page. In later chapters, you will see how the page definition files are used to define and edit the binding object definitions for specific UI components. For more information about the page definition file, see [Section 11.6, "Working with Page Definition Files"](#).

- Configures the page definition file, which includes adding definitions of the binding objects referenced by the page.
- Adds ADF Faces components to the JSF page.

These prebuilt components include ADF data binding expression language (EL) expressions that reference the binding objects in the page definition file. For more information, see [Section 11.7, "Creating ADF Data Binding EL Expressions"](#).
- Adds all the libraries, files, and configuration elements required by ADF Faces components. For more information, see the "ADF Faces Configuration" appendix in the *Oracle Fusion Middleware Web User Interface Developer's Guide for Oracle Application Development Framework*.

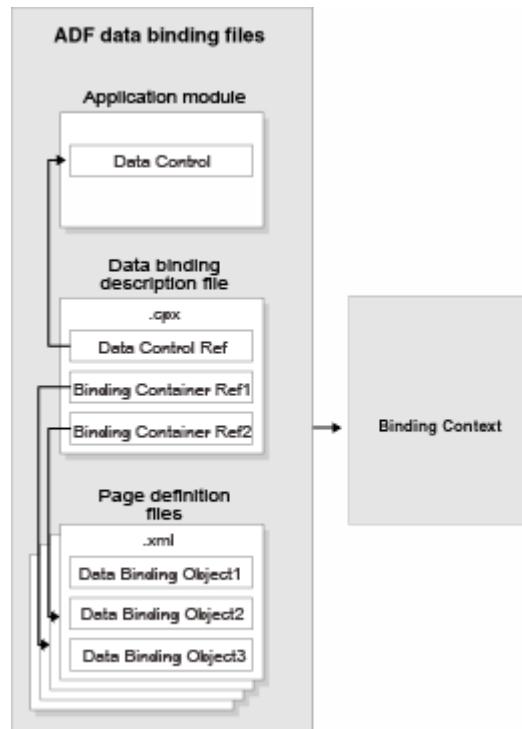
11.3.3 What Happens at Runtime: How the Binding Context Works

When a page contains ADF bindings, at runtime the interaction with the business services initiated from the client or controller is managed by the application through a single object known as the *binding context*. The binding context is a runtime map (named *data* and accessible through the EL expression `# {data}`) of all data controls and page definitions within the application.

The ADF lifecycle creates the Oracle ADF binding context from the application module, `DataBindings.cpx`, and page definition files, as shown in [Figure 11–11](#). The union of all the `DataControls.dcx` files and any application modules in the workspace define the available data controls at design time, but the `DataBindings.cpx` files define what data controls are available to the application at runtime. A `DataBindings.cpx` file lists all the data controls that are being used by pages in the application and maps the binding containers, which contain the binding objects defined in the page definition files, to web page URLs. The page definition files define the binding objects used by the application pages. There is one page definition file for each page.

The binding context does not contain real live instances of these objects. Instead, the map contains references that become data control or binding container objects on demand. When the object (such as a page definition) is released from the application, for example when a task flow ends or when the binding container or data control is released at the end of the request, data controls and binding containers turn back into reference objects. For information about the ADF lifecycle, see [Chapter 19, "Understanding the Fusion Page Lifecycle"](#).

Figure 11–11 ADF File Binding Runtime Usage



11.4 Working with the DataBindings.cpx File

The DataBindings.cpx files define the binding context for the entire application and provide the metadata from which the Oracle ADF binding objects are created at runtime. An application may have more than one DataBindings.cpx file if a component, for example a region, was created outside of the project and then imported. These files map individual pages to page definition files and declare which data controls are being used by the application. At runtime, only the data controls listed in the DataBindings.cpx files are available to the current application.

11.4.1 How JDeveloper Creates a DataBindings.cpx File

The first time you use the Data Controls panel to add a component to a page or an operation to an activity, JDeveloper automatically creates a DataBindings.cpx file in the default package of the view project. It resides in the adfmsrc directory for the project. Once the DataBindings.cpx file is created, JDeveloper adds an entry for the first page or task flow activity. Each subsequent time you use the Data Controls panel, JDeveloper adds an entry to the DataBindings.cpx for that page or activity, if one does not already exist.

Tip: JDeveloper supports refactoring. That is, you can safely rename or move many of the objects referenced in the DataBindings.cpx file, and the references will be updated. For more information, see [Chapter 30, "Refactoring a Fusion Web Application"](#).

11.4.2 What Happens When JDeveloper Creates a DataBindings.cpx File

Once JDeveloper creates a DataBindings.cpx file, you can open it in the Overview Editor. [Figure 11-12](#) shows the DataBindings.cpx file from the StoreFront module application, as viewed in the overview editor (note that it's been truncated).

Figure 11–12 DataBindings.cpx File in the Overview Editor

Data Binding Registry	
This file defines the Oracle ADF binding context for your application. JDeveloper creates this file the first time you data bind a UI component.	
Page Mappings	
path	usageId
/home.jspx	homePageDef
/templates/StoreFrontTemplate.jspx	templates_StoreFrontTemplatePageDef
/login.jspx	loginPageDef
/myOrders.jspx	myOrdersPageDef
/checkout/order.jspx	checkout_orderPageDef
/checkout/orderSummary.jspx	checkout_orderSummaryPageDef
/templates/StoreFrontMasterDetailTemplate.jspx	templates_StoreFrontMasterDetailTemplatePageDef
/WEB-INF/checkout_task_flow.smfl#checkout_task_flow#syncShoppingCartForAuthenticatedUser	WEB_INF_checkout_task_flow_checkout_task_flow_syncShoppingCartForAuthenticatedUser
/templates/StoreFrontHomeTemplate.jspx	templates_StoreFrontHomeTemplatePageDef
/account/basicInformation.jff	account_basicInformationPageDef
/account/defineAddresses.jff	account_defineAddressesPageDef
/account/updateUserInfo.jspx	account_updateUserInfoPageDef
/account/reviewCustomerInfo.jff	account_reviewCustomerInfoPageDef
/account/paymentOptions.jff	account_paymentOptionsPageDef
/account/welcomeUserRegistration.jff	account_welcomeUserRegistrationPageDef
/registerUser.jspx	registerUserPageDef
/WEB-INF/customer-registration-task-flow.xml#customer-registration-task-flow@userRegistration...	WEB_INF_customer_registration_task_flow_customer_registration_task_flow_userRegistration...
/account/summaryUserRegistration.jff	account_summaryUserRegistrationPageDef
/WEB-INF/employee-registration-task-flow.xml#employee-registration-task-flow@userRegistrati...	WEB_INF_employee_registration_task_flow_employee_registration_task_flow_userRegistratio...
/account/basicEmployeeInfo.jff	account_basicEmployeeInfoPageDef
/WEB-INF/customer-registration-task-flow.xml#customer-registration-task-flow@createAddress...	WEB_INF_customer_registration_task_flow_customer_registration_task_flow_CreateInsert...
/account/addressDetails.jff	account_addressDetailsPageDef
/account/reviewEmployeeInfo.jff	account_reviewEmployeeInfoPageDef
/checkout/editPaymentOptions.jspx	checkout_editPaymentOptionsPageDef
/checkout/addPaymentOptions.jspx	checkout_addPaymentOptionsPageDef
/checkout/addAddress.jspx	checkout_addAddressPageDef
/checkout/editAddress.jspx	checkout_editAddressPageDef
/WEB-INF/customer-registration-task-flow.xml#customer-registration-task-flow@createPayme...	WEB_INF_customer_registration_task_flow_customer_registration_task_flow_createPaymentO...
/account/paymentOptionDetails.jff	account_paymentOptionDetailsPageDef
Page Definition Usages	
id	path
homePageDef	oracle.fodemo.storefront.pageDefs.homePageDef
templates_StoreFrontTemplatePageDef	oracle.fodemo.storefront.pageDefs.templates_StoreFrontTemplatePageDef
loginPageDef	oracle.fodemo.storefront.pageDefs.loginPageDef
myOrdersPageDef	oracle.fodemo.storefront.pageDefs.myOrdersPageDef
cart_cartSummaryPageDef	oracle.fodemo.storefront.pageDefs.cart_cartSummaryPageDef
checkout_orderPageDef	oracle.fodemo.storefront.pageDefs.checkout_orderPageDef
checkout_orderSummaryPageDef	oracle.fodemo.storefront.pageDefs.checkout_orderSummaryPageDef
templates_StoreFrontMasterDetailTemplatePageDef	oracle.fodemo.storefront.pageDefs.templates_StoreFrontMasterDetailTemplatePageDef
WEB_INF_checkout_task_flow_checkout_task_flow_syncShoppingCartForAuthenticatedUser	oracle.fodemo.storefront.pageDefs.WEB_INF_checkout_task_flow_checkout_task_flow_syncS...

Example 11–1 shows an excerpt from the .cpx file in the StoreFront module application.

Example 11–1

```
<Application xmlns="http://xmlns.oracle.com/adfm/application"
    version="11.1.1.44.61" id="DataBindings" SeparateXMLFiles="false"
    Package="oracle.fodemo.storefront" ClientType="Generic">
    <definitionFactories>
        <factory nameSpace="http://xmlns.oracle.com/adf/controller/binding"
            className="oracle.adf.controller.internal.binding.
                TaskFlowBindingDefFactoryImpl"/>
        <factory nameSpace="http://xmlns.oracle.com/adfm/dvt"
            className="oracle.adfinternal.view.faces.
                dvt.model.binding.FacesBindingFactory"/>
    </definitionFactories>
    <pageMap>
        <page path="/home.jspx" usageId="homePageDef" />
        <page path="/templates/StoreFrontTemplate.jspx"
            usageId="templates_StoreFrontTemplatePageDef" />
        <page path="/login.jspx" usageId="loginPageDef" />
        <page path="/myOrders.jspx" usageId="myOrdersPageDef" />
        <page path="/checkout/order.jspx" usageId="checkout_orderPageDef" />
        .
        .
        .
        <pageDefinitionUsages>
            <page id="homePageDef"
                path="oracle.fodemo.storefront.pageDefs.homePageDef" />
            <page id="templates_StoreFrontTemplatePageDef"
                path="oracle.fodemo.storefront.pageDefs.templates_StoreFrontTemplatePageDef" />
        </pageDefinitionUsages>
    </pageMap>
</Application>
```

```

<page id="loginPageDef"
      path="oracle.fodemo.storefront.pageDefs.loginPageDef"/>
<page id="myOrdersPageDef"
      path="oracle.fodemo.storefront.pageDefs.myOrdersPageDef" />
<page id="cart_cartSummaryPageDef"
      path="oracle.fodemo.storefront.pageDefs.cart_cartSummaryPageDef" />
<page id="checkout_orderPageDef"
      path="oracle.fodemo.storefront.pageDefs.checkout_orderPageDef" />

.

.

<dataControlUsages>
  <dc id="FusionOrderDemoBookingService"
      path="orderbookingservice.FusionOrderDemoBookingService" />
  <BC4JDataControl id="StoreServiceAMDataControl"
    Package="oracle.fodemo.storefront.store.service"
    FactoryClass="oracle.adf.model.bc4j.DataControlFactoryImpl"
    SupportsTransactions="true" SupportsFindMode="true"
    SupportsRangesize="true" SupportsResetState="true"
    SupportsSortCollection="true"
    Configuration="StoreServiceAMLocal" syncMode="Immediate"
    xmlns="http://xmlns.oracle.com/adfm/datacontrol" />
  <BC4JDataControl id="ShoppingCartStoreServiceAMDataControl"
    Package="oracle.fodemo.storefront.store.service"
    FactoryClass="oracle.adf.model.bc4j.DataControlFactoryImpl"
    SupportsTransactions="true" SupportsFindMode="true"
    SupportsRangesize="true" SupportsResetState="true"
    SupportsSortCollection="true"
    Configuration="StoreServiceAMLocal" syncMode="Immediate"
    xmlns="http://xmlns.oracle.com/adfm/datacontrol" />
</dataControlUsages>
</Application>
```

The **Page Mappings** section of the editor maps each JSF page or task flow activity to its corresponding page definition file using an ID. The **Page Definition Usages** section maps the page definition ID to the absolute path for page definition file in the application. The **Data Control Usages** section identifies the data controls being used by the binding objects defined in the page definition files. These mappings allow the binding container to be initialized when the page is invoked.

You can use the overview editor to change the ID name for page definition files or data controls by double-clicking the current ID name and editing inline. Doing so will update all references in the application. Note, however, that JDeveloper updates only the ID name, it does not update the file name. Be sure that you do not change a data control name to a reserved word. For more information, see [Section 9.2.5, "How to Edit an Existing Application Module"](#).

You can also click an element in the Structure window and then use the Property Inspector to change property values. For more information about the elements and attributes in the DataBindings.cpx file, see [Section A.6, "DataBindings.cpx"](#).

Note: Do not set the Configuration property to a shared application module configuration. By default, for an application module named AppModuleName, the Property Inspector will show the configuration named AppModuleNameShared. Oracle ADF uses this special configuration automatically when you configure an application as a shared application module, but the configuration is not designed to be used by an application module data control.

11.5 Configuring the ADF Binding Filter

The ADF binding filter is a servlet filter that is an instance of the `oracle.adf.model.servlet.ADFBindingFilter` class. ADF web applications use the ADF binding filter to preprocess any HTTP requests that may require access to the binding context. To do this, the ADF binding filter must be aware of all `DataBindings.cpx` files that exist for an application.

11.5.1 How JDeveloper Configures the ADF Binding Filter

The first time you add a databound component to a page using the Data Controls panel, JDeveloper automatically configures the filter for you in the application's `web.xml` file.

11.5.2 What Happens When JDeveloper Configures an ADF Binding Filter

To configure the binding filter, JDeveloper adds the following elements to the `web.xml` file:

- An ADF binding filter class: Specifies the name of the binding filter object, which implements the `javax.servlet.Filter` interface.

The ADF binding filter is defined in the `web.xml` file, as shown in [Example 11–2](#). The `filter-name` element must contain the value `adfBindings`, and the `filter-class` element must contain the fully qualified name of the binding filter class, which is `oracle.adf.model.servlet.ADFBindingFilter`.

Example 11–2 Binding Filter Class Defined in the `web.xml` File

```
<filter>
  <filter-name>adfBindings</filter-name>
  <filter-class>oracle.adf.model.servlet.ADFBindingFilter</filter-class>
</filter>
```

- Filter mappings: Link filters to static resources or servlets in the web application.

At runtime, when a mapped resource is requested, a filter is invoked. Filter mappings are defined in the `web.xml` file, as shown in [Example 11–3](#). The `filter-name` element must contain the value `adfBindings`.

Example 11–3 Filter Mapping Defined in the `web.xml` File

```
<filter-mapping>
  <filter-name>adfBindings</filter-name>
  <servlet-name>Faces Servlet</servlet-name>
  <dispatcher>FORWARD</dispatcher>
  <dispatcher>REQUEST</dispatcher>
</filter-mapping>
```

Tip: If you have multiple filters defined in the `web.xml` file, be sure to list them in the order in which you want them to run. At runtime, the filters are executed in the sequence in which they appear in the `web.xml` file. The `adfBindings` filter should appear before any filters that depend on the ADF context to be initialized.

11.5.3 What Happens at Runtime: How the ADF Binding Filter Works

At runtime, the ADF binding filter performs the following functions:

- Overrides the character encoding when the filter is initialized with the name specified as a filter parameter in the web.xml file. The parameter name of the filter init-param element is encoding.
- Instantiates the ADFContext object, which is the execution context for a Fusion web application and contains context information about ADF, including the security context and the environment class that contains the request and response object.
- Initializes the binding context for a user's HTTP session. To do this, it first loads the bindings as defined in the DataBindings.cpx file in the current project's adfmsrc directory. If the application contains DataBindings.cpx files that were imported from another project, those files are present in the application's class path. The filter additively loads any auxiliary.cpx files found in the class path of the application.
- Serializes incoming HTTP requests from the same browser (for example, from frame sets) to prevent multithreading problems.
- Notifies data control instances that they are about to receive a request, allowing them to do any necessary per-request setup.
- Notifies data control instances after the response has been sent to the client, allowing them to do any necessary per-request cleanup.

11.6 Working with Page Definition Files

Page definition files define the binding objects that populate the data in UI components at runtime. For every page that has ADF bindings, there must be a corresponding page definition file that defines the binding objects used by that page. Page definition files provide design time access to all the ADF bindings. At runtime, the binding objects defined by a page definition file are instantiated in a binding container, which is the runtime instance of the page definition file.

Note: When multiple windows are open to the same page, the ADF Controller assigns each window its own DataControlFrame. This ensures that each window has its own binding container.

11.6.1 How JDeveloper Creates a Page Definition File

The first time you use the Data Controls panel, JDeveloper automatically creates a page definition file for that page and adds definitions for each binding object referenced by the component. For each subsequent databound component you add to the page, JDeveloper automatically adds the necessary binding object definitions to the page definition file.

By default, the page definition files are located in the view.PageDefs package in the Application Sources directory of the view project. If the corresponding JSF page is saved to a subdirectory, then the page definition will also be saved to a subdirectory in the PageDefs package. You can change the location of the page definition files using the ADFm Settings page of the Project Properties dialog.

JDeveloper names the page definition files using the following convention:

pageNamePageDef.xml

where *pageName* is the name of the JSF page. For example, if the JSF page is named *home.jsp*, the default page definition file name is *homePageDef.xml*. If you

organize your pages into subdirectories, JDeveloper prefixes the directory name to the page definition file name using the following convention:

`directoryName_pageNamePageDef.xml`

For example, in the StoreFront module, the name of the page definition file for the `updateUserInfo` page, which is in the `account` subdirectory of the Web Content node is `account_updateUserInfoPageDef.xml`.

Tip: Page definitions for task flows follow the same naming convention.

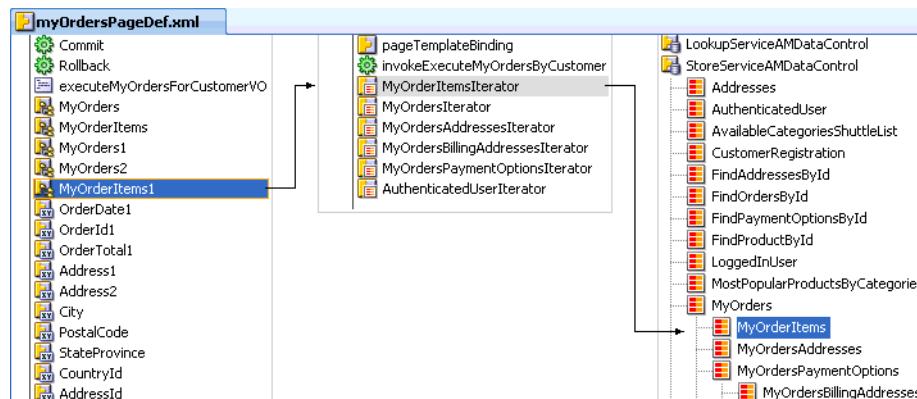
To open a page definition file, you can right-click directly on the page or activity in the visual editor, and choose **Go to Page Definition**, or for a JSF page, you can click the **Bindings** tab of the editor and click the **Page Definition File** link.

Tip: While JDeveloper automatically creates a page definition for a JSF page when you create components using the Data Controls panel, or for a task flow when you drop an item onto an activity, it does not delete the page definition when you delete the associated JSF page or task flow activity (this is to allow bindings to remain when they are needed without a JSF page). If you no longer want the page definition, you need to delete the page definition and all references to it manually. Note however, that as long as a corresponding page or activity is never called, the page definition will never be used to create a binding context. It is therefore not imperative to remove any unused page definition files from your application.

11.6.2 What Happens When JDeveloper Creates a Page Definition File

Figure 11-13 shows the page definition file in the overview editor that was created for the `myOrders.jsp` page in the StoreFront module application.

Figure 11-13 Page Definition File in the Overview Editor



The overview editor allows you to view and configure the following binding objects for a page:

- Parameters: Parameter binding objects declare the parameters that the page evaluates at the beginning of a request. (For more information about the ADF lifecycle, see [Chapter 19, "Understanding the Fusion Page Lifecycle"](#).) You can define the value of a parameter in the page definition file using static values, or EL expressions that assign a static value.

Example 11–4 shows how parameter binding objects can be defined in a page definition file.

Example 11–4 parameters Element of a Page Definition File

```
<parameters>
    <parameter id="filedBy"
        value="${bindings.userId}"/>
    <parameter id="status"
        value="${param.status != null ? param.status : 'Open'}"/>
</parameters>
```

The value of the `filedBy` parameter is defined by a binding on the `userId` data attribute, which would be an attribute binding defined later in the `bindings` element. The value of the `status` parameter is defined by an EL expression, which assigns a static value.

Tip: By default, JDeveloper uses the dollar sign (\$), which is a JSP EL syntax standard, as the prefix for EL expressions that appear in the page definition file. However, you can use the hash sign (#) prefix, which is a JSF EL syntax standard, as well.

For more information about passing parameters to methods, see [Section 26.3, "Setting Parameter Values Using a Command Component"](#).

- Model: The **Model** section of the page definition overview editor shows three different types of objects: bindings, executables, and the associated data controls (note that the data controls do not display unless you select a binding or executable). For example, in [Figure 11–13](#), you can see that the binding for the `OrderDate1` attribute uses the `MyOrdersIterator` iterator to get its value. The iterator accesses the `MyOrders` collection on the `StoreServiceAMDataControl` data control. For more information, see [Section 11.6.2.2, "Executable Binding Objects"](#).

By default, the model binding objects are named after the data control object that was used to create them. If a data control object is used more than once on a page, JDeveloper adds a number to the default binding object names to keep them unique. In [Section 11.7, "Creating ADF Data Binding EL Expressions"](#), you will see how the ADF data binding EL expressions reference the binding object names.

[Table 11–2](#) shows the icons for each of the binding objects, as displayed in the overview editor (note that while parameter objects are shown in the **Parameter** section of the editor, they are also considered binding objects).

Table 11–2 Binding Object Icons

Binding Object Type	Icon	Description
Parameter		Represents a parameter binding object.
Bindings		Represents an attribute value binding object.
		Represents a list value binding object.
		Represents a tree value binding object.

Table 11–2 (Cont.) Binding Object Icons

Binding Object Type	Icon	Description
		Represents a method action binding object
Bindings/ Executables		Represents an action binding object. Also represents an invoke action executable binding object and an event.
Executables		Represents an iterator binding object.
		Represents a task flow executable binding object.

When you click an item in the overview editor (or the associated node in the Structure window), you can use the Property Inspector to view and edit the attribute values for the item, or you can edit the XML source directly by clicking the **Source** tab.

Example 11–5 shows abbreviated XML code for the page definition file shown in Figure 11–13.

Example 11–5 Page Definition File

```
<pageDefinition xmlns="http://xmlns.oracle.com/adfm/uimodel"
    version="11.1.1.44.61" id="myOrdersPageDef"
    Package="oracle.fodemo.storefront.pageDefs"
    EnableTokenValidation="false">
<parameters/>
<executables>
    <page path="oracle.fodemo.storefront.pageDefs.
        templates_StoreFrontTemplatePageDef"
        id="pageTemplateBinding"/>
    <iterator Binds="MyOrderItems" RangeSize="25"
        DataControl="StoreServiceAMDataControl"
        id="MyOrderItemsIterator"/>
    <iterator Binds="MyOrders" RangeSize="-1"
        DataControl="StoreServiceAMDataControl" id="MyOrdersIterator"/>
    .
    .
    .
</executables>
<bindings>
    <action id="Commit" InstanceName="StoreServiceAMDataControl"
        DataControl="StoreServiceAMDataControl" RequiresUpdateModel="true"
        Action="commitTransaction"/>
    <action id="Rollback" InstanceName="StoreServiceAMDataControl"
        DataControl="StoreServiceAMDataControl" RequiresUpdateModel="false"
        Action="rollbackTransaction"/>
    <methodAction id="executeMyOrdersForCustomerVO" RequiresUpdateModel="true"
        Action="invokeMethod"
        MethodName="executeMyOrdersForCustomerVO"
        IsViewObjectMethod="false"
        DataControl="StoreServiceAMDataControl"
        InstanceName="StoreServiceAMDataControl.dataProvider"/>
    <tree IterBinding="MyOrdersIterator" id="MyOrders">
        <nodeDefinition DefName="oracle.fodemo.storefront.store.queries.OrdersVO">
            <AttrNames>
                <Item Value="OrderId"/>
                <Item Value="OrderDate"/>
            </AttrNames>
        </nodeDefinition>
    </tree>
</bindings>
```

```

        <Item Value="OrderShippedDate"/>
        <Item Value="OrderStatusCode"/>
        <Item Value="OrderTotal"/>
        <Item Value="CustomerId"/>
        <Item Value="ShipToName"/>
        <Item Value="ShipToAddressId"/>
        <Item Value="ShipToPhoneNumber"/>
        <Item Value="ShippingOptionId"/>
        <Item Value="PaymentOptionId"/>
        <Item Value="CalculatedOrderTotal"/>
        <Item Value="TotalShippingCost"/>
        <Item Value="DiscountAmount"/>
        <Item Value="InvoiceTotal"/>
        <Item Value="LastUpdateDate"/>
        <Item Value="TypedCouponCode"/>
    </AttrNames>
</nodeDefinition>
</tree>
<attributeValues IterBinding="MyOrdersIterator" id="OrderDate1">
    <AttrNames>
        <Item Value="OrderDate"/>
    </AttrNames>
</attributeValues>
<attributeValues IterBinding="MyOrdersIterator" id="OrderId1">
    <AttrNames>
        <Item Value="OrderId"/>
    </AttrNames>
</attributeValues>
.
.
.
</bindings>
</pageDefinition>

```

In later chapters, you will see how the page definition file is used to define and edit the bindings for specific UI components. For a description of all the possible elements and attributes in the page definition file, see [Section A.7, "pageNamePageDef.xml"](#).

11.6.2.1 Bindings Binding Objects

There are three types of Bindings binding objects used to bind UI components to objects on the data control:

- **Value:** Displays data in UI components by referencing an iterator binding. Each discrete UI component on a page that will display data from the data control is bound to a value binding object. Value binding objects include:
 - **Attribute Values:** Binds text fields to a specific attribute in an object (also referred to as an *attribute binding* object.)
 - **List:** Binds the list items to all values of an attribute in a data collection.
 - **Tree:** Binds an entire table to a data collection and can also bind the root node of a tree to a data collection.
 - **Button (boolean):** Binds a checkbox to a boolean value for an attribute.
 - **Graph:** Binds a graph directly to the source data.
- **Method Action:** Binds command components, such as buttons or links, to custom methods on the data control. A method action binding object encapsulates the

- details about how to invoke a method and what parameters (if any) the method is expecting.
- Action: Binds command components, such as buttons or links, to built-in data control operations (such as, Commit or Rollback) or to built-in collection-level operations (such as, Create, Delete, Next, or Previous).

Collectively, the binding objects are referred to as *control* binding objects, because they work with the UI controls on a page.

[Example 11–6](#) shows a sample bindings element, which defines one action binding called Commit, one attribute binding for a text field called PaymentOptionID1, and one list binding called PaymentTypeCode.

Example 11–6 **bindings Element of a Page Definition File**

```
<bindings>
    <action id="Commit" InstanceName="StoreServiceAMDataControl"
        DataControl="StoreServiceAMDataControl" RequiresUpdateModel="true"
        Action="commitTransaction"/>
    <attributeValues IterBinding="PaymentOptionsForUserIterator"
        id="PaymentOptionId1">
        <AttrNames>
            <Item Value="PaymentOptionId"/>
        </AttrNames>
    </attributeValues>
    <list IterBinding="PaymentOptionsForUserIterator" id="PaymentTypeCode"
        Uses="LOV_PaymentTypeCode" StaticList="false">
        <AttrNames>
            <Item Value="PaymentTypeCode"/>
        </AttrNames>
    </list>
</bindings>
```

The binding object defined in the action element encapsulates the information needed to invoke the built-in commit operation on the StoreServiceAMDataControl data control. The value of true in the RequiresUpdateModel attribute specifies that the model layer needs to be updated before the operation is executed.

If this operation also raised a contextual event, an event definition would appear well. If the page contained bindings that consumed an event, the event mapping would also appear. For more information, see [Section 26.5, "Creating Contextual Events"](#).

The attributeValues element defines the value bindings for the text fields on the page. In the example, the PaymentOptionId1 attribute binding will display the value of the PaymentOptionId, which is defined in the AttrNames element. The IterBinding attribute references the iterator binding that manages the data to be displayed in the text field (for more information, see [Section 11.6.2.2, "Executable Binding Objects"](#)).

The PaymentTypeCode element defines the list binding used to display the list of payment type codes by accessing the LOV created on the PaymentOptions view object. For more information about creating lists using LOVs on view objects, see [Chapter 23, "Creating Databound Selection Lists and Shuttles"](#).

11.6.2.2 Executable Binding Objects

There are seven types of executable binding objects:

- **Iterator:** Binds to an iterator that iterates over view object collections. There is one iterator binding for each collection used on the page. All of the value bindings on the page must refer to an iterator binding in order for the component values to be populated with data at runtime.

When you drop a collection or an attribute of a collection on the page, an iterator binding is automatically added as an executable. Iterator binding objects bind to an underlying ADF RowSetIterator object, which manages the current object and current range information. The iterator binding exposes the current object and range state to the other binding objects used by the page. The iterator *range* represents the current set of objects to be displayed on the page. The maximum number of objects in the current range is defined in the `rangeSize` attribute of the iterator. For example, if a collection in the data control contains products and the iterator range size is 25, the first 25 products in the collection are displayed on the page. If the user scrolls down, the next set of 25 is displayed, and so on. If the user scrolls up, the previous set of 25 is displayed. If your view object uses range paging, then you can configure the iterator binding to return a set of ranges at one time. For more information, see [Section 35.1.5, "Efficiently Scrolling Through Large Result Sets Using Range Paging"](#).

Note: If you have two pages each with an iterator binding bound to the iterator on the same view object (which you will if you drop the same collection, for example, on two different pages), then you should ensure that the `rangeSize` attribute is the same for both pages' iterator bindings. If not, the page with a smaller range size may cause the iterator to reexecute, causing unexpected results on the other page.

- **Method Iterator:** Binds to an iterator that iterates over the collections returned by custom methods in the data control.

A method iterator binding is always related to a method action binding object. The method action binding encapsulates the details about how to invoke the method and what parameters (if any) the method is expecting. The method action binding is itself bound to the method iterator, which provides the data.

You will see method iterator executable binding objects only if you drop a method return collection or an attribute of a method return collection from a custom method on the data control. If you are using only application module data controls, you will see only iterator binding objects.

- **Variable Iterator:** Binds to an iterator that exposes all the variables in the binding container to the other bindings. While there is an iterator binding for each collection, there is only one variable iterator binding for all variables used on the page. (The variable iterator is like an iterator pointing to a collection that contains only one data object whose attributes are the binding container variables.)

Page variables are local to the binding container and exist only while the binding container object exists. When you use a data control method (or an 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 page variables.

A variable iterator can contain one of two types of variables: `variable` and `variableUsage`. A `variable` type variable is a simple value holder, while a `variableUsage` type variable is a value holder that is related to a view object's named bind parameter. Defining a variable as a `variableUsage` type allows it to

- inherit the default value and UI control hints from the view object named bind variable to which it is bound.
- Invoke Action: Binds to a method that invokes the operations or methods defined in action or method action bindings during any phase of the page lifecycle.

Tip: If you know you want a method to execute before the page is rendered, you should use a method call activity in the task flow to invoke the method, rather than an invoke action in the page definition file. Using the method call activity makes invoking page logic easier, and allows you to show more information on the task flow, making the diagram more readable and useful to anyone else who might be using it. However, if you need the method to be executed in more than one phase of the page's lifecycle, or if you plan to reuse the page and page definition file and want the method to be tied to the page, or if your application does not use ADFc, then you should use an invoke action to invoke the method.
- Page: Binds to the template's page definition file (if a template is used). For more information about how this works with templates, see [Section 18.2, "Using Page Templates"](#).

Note: You can also use the page element to bind to another page definition file. However, at runtime, only the current incoming page's (or if the rendered page is different from the incoming, the rendered page's) binding container is automatically prepared by the framework during the current request. Therefore, to successfully access a bound value in another page from the current page, you must programmatically prepare that page's binding container in the current request (for example, using a backing bean). Otherwise, the bound values in that page may not be available or valid in the current request.

- Search Region: Binds named criteria to the iterator, so that the search can be executed.
- Task Flow: Instantiates the binding container for a region's task flow.

At runtime, executable bindings are refreshed based on the value of their Refresh attribute. Refreshing an iterator binding reconnects it with its underlying RowSetIterator object. Refreshing an invoke action binding invokes the action. Before refreshing any bindings, the ADF runtime evaluates any Refresh and RefreshCondition attributes specified in the executables. The Refresh attribute specifies the ADF lifecycle phase within which the executable should be invoked. The RefreshCondition attribute specifies the conditions under which the executable should be invoked. You can specify the RefreshCondition value using a boolean EL expression. If you leave the RefreshCondition attribute blank, it evaluates to true.

By default, the Refresh value is set to deferred. This means the binding will not be executed unless its value is accessed (for example by an EL expression on a JSF page). Once called, it will not re-execute unless any parameter values for the binding have changed, or if the binding itself has changed.

For more information about how bindings are refreshed and how to set the Refresh and RefreshCondition attributes, see [Section 19.2, "The JSF and ADF Page Lifecycles"](#).

[Example 11–7](#) shows an example of executable binding objects.

Example 11–7 executable Binding Objects in a Page Definition File

```
<executables>
  <page path="oracle.fodemo.storefront.pageDefs.
    templates_StoreFrontTemplatePageDef"
        id="pageTemplateBinding" />
  <iterator Binds="MyOrderItems" RangeSize="25"
            DataControl="StoreServiceAMDataControl"
            id="MyOrderItemsIterator"/>
  <iterator Binds="MyOrders" RangeSize="-1"
            DataControl="StoreServiceAMDataControl" id="MyOrdersIterator" />
```

The invokeAction object invokes the executeMyOrdersForCustomerVO method, which is named in the Binds attribute. In the bindings wrapper element, the methodAction binding object encapsulates the details about how to invoke the method.

The iterator binding named MyOrderItemsIterator was created by dropping the MyOrderItem collection on the page as a table. The iterator binding named MyOrdersIterator was created by dropping the MyOrders collection, which has a master-detail relationship with the MyOrderItems collection. For more information, see [Chapter 22, "Displaying Master-Detail Data"](#).

The Binds attribute of the iterator element defines the collection the iterator will iterate over. The RangeSize attribute defines the number of objects the iterator is to display on the page at one time. A RangeSize value of -1 causes the iterator to display *all* the objects from the collection.

Tip: Normally, an iterator binding's default range size is 25. However, when an iterator binding is created from the Edit List Binding dialog, the range size defaults to -1 so that all choices display in the list, not just the first 25.

Performance Tip: When you want to reduce the number of roundtrips the iterator requires to fetch the data objects from the view object in the Business Components layer, you can set the rangeSize attribute to -1, and the objects will be fetched in a single round trip to the server, rather than in multiple trips as the user navigates through the objects.

11.7 Creating ADF Data Binding EL Expressions

To display data from the data model, web page UI components are bound to binding objects using JSF Expression Language (EL) expressions. These EL expressions reference a specific binding object in a binding container. At runtime, the JSF runtime evaluates an EL expression and pulls the value from the binding object to populate the component with data when the page is displayed. If the user updates data in the UI component, the JSF runtime pushes the value back into the corresponding binding object based on the same EL expression.

Tip: There may be cases when you need to use EL expressions within managed beans. For information on working with EL expressions within managed beans, see the "Creating EL Expressions" section in the *Oracle Fusion Middleware Web User Interface Developer's Guide for Oracle Application Development Framework*.

11.7.1 How to Create an ADF Data Binding EL Expression

When you use the Data Controls panel to create a component, the ADF data binding expressions are created for you. The expressions are added to every component attribute that will either display data from or reference properties of a binding object. Each prebuilt expression references the appropriate binding objects defined in the page definition file. You can edit these binding expressions or create your own, as long as you adhere to the basic ADF binding expression syntax. ADF data binding expressions can be added to any component attribute that you want to populate with data from a binding object.

In JSF pages, a typical ADF data binding EL expression uses the following syntax to reference any of the different types of binding objects in the binding container:

```
#{bindings.BindingObject.propertyName}
```

where:

- *bindings* is a variable that identifies that the binding object being referenced by the expression is located in the binding container of the current page. All ADF data binding EL expressions must start with the *bindings* variable.
- *BindingObject* is the ID, or for attributes the name, of the binding object as it is defined in the page definition file. The binding object ID or name is unique to that page definition file. An EL expression can reference any binding object in the page definition file, including parameters, executables, or value bindings.
- *propertyName* is a variable that determines the default display characteristics of each databound UI component and sets properties for the binding object at runtime. There are different binding properties for each type of binding object. For more information about binding properties, see [Section 11.7.2, "What You May Need to Know About ADF Binding Properties"](#).

For example, in the following expression that might appear on a JSF page:

```
#{bindings.ProductName.inputValue}
```

the *bindings* variable references a bound value in the current page's binding container. The binding object being referenced is *ProductName*, which is an attribute binding object. The binding property is *inputValue*, which returns the value of the first *ProductName* attribute.

Tip: While the binding expressions in the page definition file can use either a dollar sign (\$) or hash sign (#) prefix, the EL expressions in JSF pages can use only the hash sign (#) prefix.

As stated previously, when you use the Data Controls panel to create UI components, these expressions are built for you. However, you can also manually create them if you need to. The JDeveloper Expression Builder is a dialog that helps you build EL expressions by providing lists of binding objects defined in the page definition files, as well as other valid objects to which a UI component may be bound. It is particularly useful when creating or editing ADF databound expressions because it provides a hierarchical list of ADF binding objects and their most commonly used properties. For

information about binding properties, see [Section 11.7.2, "What You May Need to Know About ADF Binding Properties"](#).

11.7.1.1 Opening the Expression Builder from the Property Inspector

You can select an item in the visual editor, and then create EL expressions for specific attributes using the Property Inspector.

To open the Expression Builder from the Property Inspector:

1. Select a UI component in the Structure window or the visual editor.
2. In the Property Inspector, click the dropdown list next to a field, and choose **Expression Builder**.

11.7.1.2 Using the Expression Builder

Once the Expression Builder is open, you can use it to create EL expressions.

To use the Expression Builder:

1. Open the Expression Builder dialog.
2. Use the Expression Builder to edit or create ADF binding expressions using the following features:
 - Use the **Variables** tree to select items that you want to include in the binding expression. The tree contains a hierarchical representation of the binding objects. Each icon in the tree represents various types of binding objects that you can use in an expression (see [Table 11–3](#) for a description of each icon). To narrow down the tree, you can either use the dropdown filter or enter search criteria in the search field. Double-click an item in the tree to move it to the **Expression** box.
 - You can also type the expression directly in the **Expression** box. JDeveloper provides code insight in the Expression Builder. To invoke code insight, type the leading characters of an EL expression (for example, # {) or a period separator. Code insight displays a list of valid items for each segment of the expression from which you can select the one you want.
 - Use the operator buttons to add logical or mathematical operators to the expression.

Table 11–3 Icons Under the ADF Bindings Node of the Expression Builder

Icon	Description
 bindings	Represents the <code>bindings</code> container variable, which references the binding container of the current page. Opening the <code>bindings</code> node exposes all the binding objects for the current page.
 data	Represents the <code>data</code> binding variable, which references the entire binding context (created from all the <code>.cpx</code> files in the application). Opening the <code>data</code> node exposes all the page definition files in the application.
	Represents an action binding object. Opening a node that uses this icon exposes a list of valid action binding properties.
	Represents an iterator binding object. Opening a node that uses this icon exposes a list of valid iterator binding properties.

Table 11-3 (Cont.) Icons Under the ADF Bindings Node of the Expression Builder

Icon	Description
	Represents an attribute binding object. Opening a node that uses this icon exposes a list of valid attribute binding properties.
	Represents a list binding object. Opening a node that uses this icon exposes a list of valid list binding properties.
	Represents a table or tree binding object. Opening a node that uses this icon exposes a list of valid table and tree binding properties.
	Represents an ADF binding object property. For more information about ADF properties, see Section 11.7.2, "What You May Need to Know About ADF Binding Properties" .
	Represents a parameter binding object.
	Represents a JavaBean.
	Represents a method.

11.7.2 What You May Need to Know About ADF Binding Properties

When you create a databound component using the Expression Builder, the EL expression might reference specific ADF binding properties. At runtime, these binding properties can define such things as the default display characteristics of a databound UI component or specific parameters for iterator bindings. The ADF binding properties are defined by Oracle APIs. For a full list of the available properties for each binding type, see [Appendix B, "Oracle ADF Binding Properties"](#).

Values assigned to certain properties are defined in the page definition file. For example, iterator bindings have a property called `RangeSize`, which specifies the number of rows the iterator should display at one time. The value assigned to `RangeSize` is specified in the page definition file, as shown in [Example 11-8](#).

Example 11-8 Iterator Binding Object with the RangeSize Property

```
<iterator Binds="ProductsByCategory1" RangeSize="25"
          DataControl="StoreFrontModuleDataControl"
          id="Products2Iterator"/>
```

11.8 Using Simple UI First Development

While the Data Controls panel enables you to design and create bound components in a single drag-and-drop action, in some cases, it may be preferable to create the basic UI components first and add the bindings later. For example, if your page will use declarative components, you will first need to drop the declarative component, and then bind it to the correct ADF control. Declarative components are reusable, composite UI components that are made up of other ADF Faces components. Once imported into a project, declarative components can be dropped onto a page from the Component Palette, similar to standard ADF Faces components. While the entire declarative component cannot use ADF data binding, you can use ADF data binding on the individual components that make up the declarative component, once the declarative component is dropped on the page. For more information about declarative components, see the "Using Declarative Components in JSF Pages" section

of the *Oracle Fusion Middleware Web User Interface Developer's Guide for Oracle Application Development Framework*.

Note: If you know the UI components on your page will eventually use ADF data binding, but you need to develop the pages before the data controls are ready, then you should consider using placeholder data controls, rather than manually binding the components. Using placeholder data controls will provide the same declarative development experience as using developed data controls. For more information, see [Chapter 27, "Designing a Page Using Placeholder Data Controls"](#).

When designing web pages, keep in mind that ADF bindings can be added only to certain ADF Faces tags or their equivalent JSF HTML tags. [Table 11–4](#) lists the ADF Faces and JSF tags to which you can later add ADF bindings.

Tip: To enable the use of JSF Reference Implementation UI component tags with ADF bindings, you must choose the **Include JSF HTML Widgets for JSF Databinding** option in the **ADF View Settings** of the project properties. However, using ADF Faces tags, especially with ADF bindings, provides greater functionality than does using the reference implementation JSF tags.

Table 11–4 Tags That Can Be Used for ADF Bindings

ADF Faces Tags Used in ADF Bindings	Equivalent JSF HTML Tags
Text Fields	
af:inputText	h:inputText
af:outputText	h:outputText
af:outputLabel	h:outputLabel
af:inputDate	n/a
Tables	
af:table	h:dataTable
Actions	
af:commandButton	h:commandButton
af:commandLink	h:commandLink
af:commandMenuItem	n/a
af:commandToolbarButton	n/a
Selection Lists	
af:inputListOfValues	n/a
af:selectOneChoice	h:selectOneMenu
af:selectOneListbox	h:selectOneListbox
af:selectOneRadio	h:selectOneRadio
af:selectBooleanCheckbox	h:selectBooleanCheckbox
Queries	
af:query	n/a

Table 11–4 (Cont.) Tags That Can Be Used for ADF Bindings**ADF Faces Tags Used in ADF Bindings Equivalent JSF HTML Tags**

af:quickQuery	n/a
Trees	
af:tree	n/a
af:treeTable	n/a

Before adding binding to the UI components, ensure that you follow these guidelines:

- When creating the JSF page using the Create JSF JSP wizard, choose the **Do not Automatically Expose UI Components in a Managed Bean** option in the **Page Implementation** section.

This option turns off JDeveloper's auto-binding feature, which automatically associates every UI component in the page to a corresponding property in the backing bean for eventual programmatic manipulation. If you intend to add ADF bindings to a page, do not use the auto-binding feature. If you use the auto-binding feature, you will have to remove the managed bean bindings later, after you have added the ADF bindings. The managed bean UI component property bindings do not affect the ADF bindings, but their presence may be confusing in the JSF code. For information about managed beans, see [Section 18.4, "Using a Managed Bean in a Fusion Web Application"](#).

- Add the ADF Faces tag libraries.

While you can add ADF bindings to JSF components, the ADF Faces components provide greater functionality, especially when combined with ADF bindings.

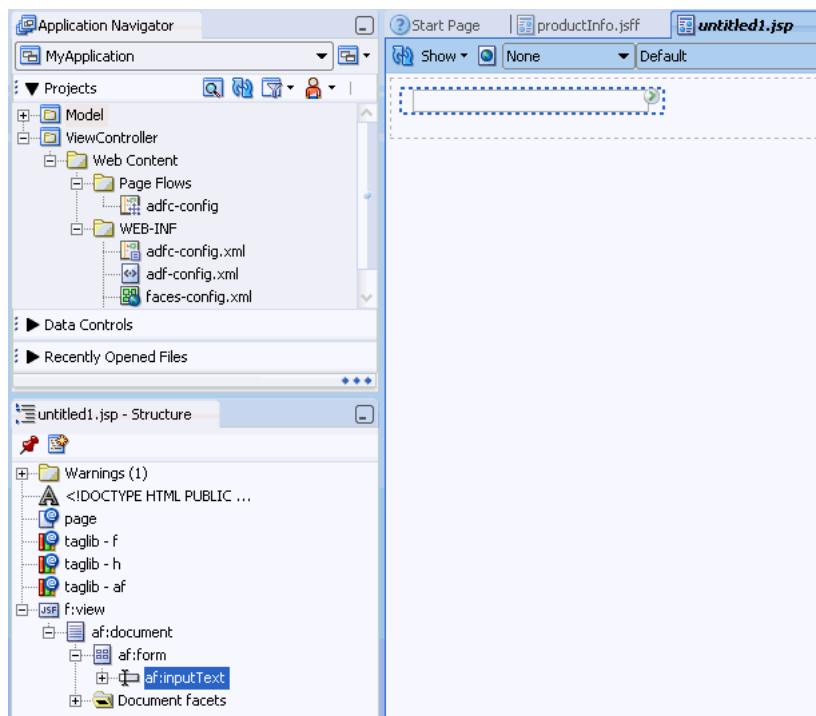
11.8.1 How to Apply ADF Model Data Binding to Existing UI Components

You apply ADF model binding to components using the Structure window.

To apply ADF Model data binding:

1. In the Design page of the visual editor, select the UI component to which you want to add ADF bindings.

The component must be one of the tags listed in [Table 11–4](#). When you select a component in the visual editor, JDeveloper simultaneously selects that component tag in the Structure window, as shown in [Figure 11–14](#).

Figure 11–14 The Structure Window in JDeveloper

2. In the Structure window, right-click the UI component, and from the context menu, choose **Bind to ADF Control**.

Note: Your project must already contain data controls for the **Bind to ADF Control** menu option to appear. If yours does not, you should consider using placeholder data controls, as described in [Chapter 27, "Designing a Page Using Placeholder Data Controls"](#).

3. In the Bind to ADF Control dialog, select the data control to which you want the UI component bound. JDeveloper will notify you if you choose a control that is not compatible with the selected UI component.

11.8.2 What Happens When You Apply ADF Model Data Binding to UI Components

When you use the Data Controls panel all of the required ADF objects are automatically created for you, as described in [Section 11.3.2, "What Happens When You Use the Data Controls Panel"](#).

12

Integrating Web Services Into a Fusion Web Application

This chapter describes how to call a third-party web service and work directly with the service proxy and service data objects (SDOs) programmatically for all common remote service data access tasks and also how to create ADF data controls for third-party web services when you want to work with the web service in the user interface.

This chapter includes the following sections:

- [Section 12.1, "Introduction to Web Services in Fusion Web Applications"](#)
- [Section 12.2, "Calling a Web Service from an Application Module"](#)
- [Section 12.3, "Creating Web Service Data Controls"](#)
- [Section 12.4, "Securing Web Service Data Controls"](#)

12.1 Introduction to Web Services in Fusion Web Applications

Web services allow enterprises to expose business functionality irrespective of the platform or language of the originating application because the business functionality is exposed in such a way that it is abstracted to a message composed of standard XML constructs that can be recognized and used by other applications.

Web services are modular business services that can be easily integrated and reused, and it is this that makes them ideally suited as components within SOA. JDeveloper helps you to create top-down web services (services created starting from a WSDL), bottom-up web services (created from the underlying implementation such as a Java class or a PL/SQL stored procedure in a database), and services created from existing functionality such as exposing an application module as a service.

You can consume web services in web applications, and common reasons for wanting to do so are:

- To add functionality which would be time-consuming to develop with the application, but which is readily available as a web service
- To access an application that runs on different architecture
- To access an application that is owned by another team when their application must be independently installed, upgraded, and maintained, especially when its data is not replicated locally (for example, when other methods of accessing their application cannot be used)

12.2 Calling a Web Service from an Application Module

In a service-oriented architecture, your Oracle ADF application module may need to take advantage of functionality offered by a web service that is not based on an application module. A web service can be implemented in any programming language and can reside on any server on the network. Each web service identifies the methods in its API by describing them in a standard, language-neutral XML format. This XML document, whose syntax adheres to the Web Services Description Language (WSDL), enables JDeveloper to understand the names of the web service's methods, as well as the data types of the parameters they might expect and their eventual return value.

JDeveloper's built-in web services wizards make this an easy task. Create a web service proxy class using the wizard, then call the service using method calls you add to a local Java object.

12.2.1 How to Call an External Service Programmatically

To call a web service from an application module, you create a web service proxy class for the service you want to invoke. A web service proxy is a generated Java class that represents the web service inside your application. It encapsulates the service URL of the web service and handles the lower-level details of making the call.

To work with a web service, you need to know the URL that identifies its WSDL document. If you have received the WSDL document as an email attachment, for example, and saved it to your local hard drive, the URL could be similar to:

`file:///D:/temp/SomeService.wsdl`

Alternatively, the URL could be an HTTP-based URL like:

`http://someserver.somecompany.com/SomeService/SomeService.wsdl`

Some web services make their WSDL document available by using a special parameter to modify the service URL. For example, a web service that expects to receive requests at the HTTP address of `http://someserver.somecompany.com/SomeService` might publish the corresponding WSDL document using the same URL with an additional parameter on the end, like this:

`http://someserver.somecompany.com/SomeService?WSDL`

Since there is no established standard, you will just need to know what the correct URL to the WSDL document is. With the URL information, you can then create a web service proxy class to call the service.

ADF Business Components services have URLs to the service of the following formats:

- On Integrated WLS, the URL has the format
`http://host:port/EJB-context-root/@WebService-name?WSDL`,
for example:

`http://localhost:8888/EJB-StoreFrontService/StoreFrontService?WSDL`

- On Oracle Application Server, the URL has the format
`http://host:port/context-root/@WebService-name?WSDL`,
for example:

`http://localhost:8888/StoreFrontService/StoreFrontService?WSDL`

The web service proxy class presents a set of Java methods that correspond to the web service's public API. By using the web service proxy class, you can call any method in

the web service in the same way as you work with the methods of any other local Java class.

To call a web service from an application module using a proxy class, you perform the following tasks:

1. Create a web service proxy class for the web service. To create a web service proxy class for a web service you need to call, use the Create Web Service Proxy wizard.
2. Implement the methods in the proxy class to access the desired web services.
3. Create an instance of the web service proxy class in your application module and invoke one or more methods on the web service proxy object.

12.2.1.1 Creating a Web Service Proxy Class to Programmatically Access the Service

To create a web service proxy class for a web service you need to call, use the Create Web Service Proxy wizard.

To create a web service proxy class to programmatically access the service:

1. In the Application Navigator, right-click the project in which you want to create the web service proxy, and choose **New**.
2. In the New Gallery, expand **Business Tier**, select **Web Services**, and then select **Web Service Proxy**, and click **OK**.
3. On the Select Web Service Description page of the wizard, enter or choose a Java package name for the generated web service proxy class.
4. Enter the URL for the WSDL of the service you want to call in your application, and then tab out of the field.

If the **Next** button does not enable, click **Why Not?** to understand what problem JDeveloper encountered when trying to read the WSDL document. If necessary, fix the problem after verifying the URL and repeat this step.

5. When the wizard displays **Next** enabled, then JDeveloper has recognized and validated the WSDL document. You can click **Next** and continue.
6. Click **Finish**.

12.2.1.2 Calling the Web Service Proxy Template to Invoke the Service

After you create the web service proxy, you must Implement the methods in the proxy class to access the desired web services.

To call the web service proxy template to invoke the service:

1. Open the proxy client class, called *port_nameClient.java*, in the source editor, and locate the comment `// Add your own code here`, which is in a try-catch block in the main method.
2. Add the appropriate code to invoke the web service.
3. Deploy the full set of client module classes that JDeveloper has generated, and reference this class in your application.

12.2.1.3 Calling a Web Service Method Using the Proxy Class in an Application Module

After you've generated the web service proxy class, you can use it inside a custom method of your application module, as shown in [Example 12-1](#). The method creates an

instance of the web service proxy class and calls the web service method from the web service proxy class for the result.

Example 12–1 Web Service Proxy Class Calls Web Service Method

```
// In YourModuleImpl.java
public void performSomeApplicationTask(String symbol) throws Exception {
    // application-specific code here
    :
    // Create an instance of the web service proxy class
    StockQuoteServiceSoapHttpPortClient svc =
        new StockQuoteServiceSoapHttpPortClient();
    // Call a method on the web service proxy class and get the result
    QuoteInfo quote = svc.quoteForSymbol(symbol);
    float currentPrice = quote.getPrice();
    // more application-specific code here
}
```

12.2.2 What Happens When You Create the Web Service Proxy

JDeveloper generates the web service proxy class in the package you've indicated with a name that reflects the name of the web service port it discovered in the WSDL document. The web service port name might be a human-readable name like StockQuoteService, or could be a less-friendly name like StockQuoteServiceSoapHttpPort. The port name is decided by the developer that published the web service you are using. If the port name of the service were StockQuoteServiceSoapHttpPort, for example, JDeveloper would generate a web proxy class named StockQuoteServiceSoapHttpPortClient.

The web service proxy displays in the Application Navigator as a single, logical node called *WebServiceNameProxy*. For example, the node for the StockQuoteService web service above would appear in the navigator with the name StockQuoteServiceProxy. As part of generating the proxy class, in addition to the main web service proxy class that you use to invoke the server, JDeveloper generates a number of auxiliary classes and interfaces. You can see these files in the Application Navigator under the *WebServiceNameProxy* node. The generated files are used as part of the lower-level implementation of invoking the web service.

The only auxiliary generated classes you need to reference are those created to hold structured web service parameters or return types. For example, imagine that the StockQuoteService web service has a `quoteForSymbol()` method that accepts one `String` parameter and returns a floating-point value indicating the current price of the stock. If the designer of the web service chose to return a simple floating-point number, then the web service proxy class would have a corresponding method like this:

```
public float quoteForSymbol(String symbol)
```

If instead the designer of the web service thought it useful to return multiple pieces of information as the result, then the service's WSDL file would include a named structure definition describing the multiple elements it contains. For example, assume that the service returns both the symbol name and the current price as a result. To contain these two data elements, the WSDL file might define a structure named `QuoteInfo` with an element named `symbol` of `string` type and an element named `price` of floating-point type. In this situation, when JDeveloper generates the web service proxy class, the Java method signature would look like this instead:

```
public QuoteInfo quoteForSymbol(String symbol)
```

The `QuoteInfo` return type references one of the auxiliary classes that comprises the web service proxy implementation. It is a simple bean whose properties reflect the names and types of the structure defined in the WSDL document. In a similar way, if the web service accepts parameters whose values are structures or arrays of structures, then you will work with these structures in your Java code using the corresponding generated beans.

12.2.3 What Happens at Runtime: When You Call a Web Service Using a Web Service Proxy Class

When you invoke a web service from an application module, the web service proxy class handles the lower-level details of using the XML-based web services protocol described in SOAP. In particular, it does the following:

- Creates an XML document to represent the method invocation
- Packages any method arguments in XML
- Sends the XML document to the service URL using an HTTP POST request
- Unpackages the XML-encoded response from the web service

If the method you invoke has a return value, your code receives it as an appropriately typed object to work with in your application module code.

12.2.4 What You May Need to Know About Web Service Proxies

12.2.4.1 Using a Try-Catch Block to Handle Web Service Exceptions

By using the generated web service proxy class, invoking a remote web service becomes as easy as calling a method in a local Java class. The only distinction to be aware of is that the web service method call could fail if there is a problem with the HTTP request involved. The method calls that you perform against a web service proxy should anticipate the possibility that the request might fail by wrapping the call with an appropriate `try...catch` block. [Example 12–2](#) improves on the simpler example (shown in [Section 12.2.1.3, "Calling a Web Service Method Using the Proxy Class in an Application Module"](#)) by catching the web service exception. In this case, it simply rethrows the error as a `JboException`, but you could implement more appropriate error handling in your own application.

Example 12–2 Wrapping Web Service Method Calls with a Try-Catch Block

```
// In YourModuleImpl.java
public void performSomeApplicationTask(String symbol) {
    // application-specific code here
    //
    QuoteInfo quote = null;
    try {
        // Create an instance of the web service proxy class
        StockQuoteServiceSoapHttpPortClient svc =
            new StockQuoteServiceSoapHttpPortClient();
        // Call a method on the web service proxy class and get the result
        quote = svc.quoteForSymbol(symbol);
    }
    catch (Exception ex) {
        throw new JboException(ex);
    }
    float currentPrice = quote.getPrice();
```

```
// more application-specific code here  
}
```

12.2.4.2 Separating Application Module and Web Services Transactions

You will use some web services to access reference information. However, other services you call may modify data. This data modification might be in your own company's database if the service was written by a member of your own team or another team in your company. If the web service is outside your firewall, of course the database being modified will be managed by another company.

In either of these situations, it is important to understand that any data modifications performed by a web service you invoke will occur in their own distinct transaction, unrelated to the application module's current unit of work. For example, if you have invoked a web service that modifies data and then you later call `rollback()` to cancel the pending changes in the application module's current unit of work, this has no effect on the changes performed by the web service you called in the process. You may need to invoke a corresponding web service method to perform a compensating change to account for your rollback of the application module's transaction.

12.2.4.3 Setting Browser Proxy Information

If the web service you need to call resides outside your corporate firewall, you need to ensure that you have set the appropriate Java system properties to configure the use of an HTTP proxy server. The Java system properties to configure are:

- `http.proxyHost` — Set this to the name of the proxy server.
- `http.proxyPort` — Set this to the HTTP port number of the proxy server (often 80).
- `http.nonProxyHosts` — Optionally set this to a vertical-bar-separated list of servers not requiring the use of a proxy server (for example, `localhost | 127.0.0.1 | *.yourcompany.com`).

Within JDeveloper, you can configure an HTTP proxy server on the Web Browser and Proxy page of the Preferences dialog. When you run your application, JDeveloper includes appropriate `-D` command-line options to set these three system properties based on the settings you've indicated in this dialog.

12.2.4.4 Invoking Application Modules with a Web Service Proxy Class

If you use a web service proxy class to invoke an Oracle ADF service-based application module, you lose the ability to optimize the call when the calling component and the service you are calling are colocated.

12.3 Creating Web Service Data Controls

The most common way of using web services in an application developed using Oracle ADF is to create a data control for an external web service. A typical reason for doing this is to add functionality that is readily available as a web service, but which would be time consuming to develop with the application, or to access an application that runs on a different architecture.

Additionally, you can reuse components created by Oracle ADF to make them available as web services for other applications to access.

12.3.1 How to Create a Web Service Data Control

JDeveloper allows you to create a data control for an existing web service using just the WSDL for the service. You can browse to a WSDL on the local file system, locate one in a UDDI registry, or enter the WSDL URL directly.

Note: If you are working behind a firewall and you want to use a web service that is outside the firewall, you must configure the Web Browser and Proxy settings in JDeveloper. For more information, see [Section 12.2.4.3, "Setting Browser Proxy Information"](#).

To create a web service data control:

1. In the Application Navigator, right-click an application and choose **New**.
2. In the New Gallery, expand **Business Tier**, select **Web Services**, and then select **Web Service Data Control**, and click **OK**.
3. Follow the wizard instructions to complete creating the data control.

12.3.2 How to Adjust the Endpoint for a Web Service Data Control

After developing a web service data control, you can modify the endpoint. This is useful, for example, when you migrate the application from a test environment to production.

To change the endpoint for a web service data control:

1. In the Application Navigator, select the .dcx file for the web service data control.
2. In the Structure window, right-click the web service data control and choose **Edit Web Service Connection** from the context menu.
3. In the Edit Web Service Connection dialog, make the necessary changes to the endpoint URL and port name.
4. Click **OK**.

12.3.3 What You May Need to Know About Web Service Data Controls

As with other kinds of data controls, you can design a databound user interface by dragging an item from the Data Controls panel and dropping it on a page as a specific UI component. For more information, see [Section 11.3.1, "How to Use the Data Controls Panel"](#).

In the Data Controls panel, each data control object is represented by an icon. [Table 12-1](#) describes what each icon represents, where it appears in the Data Controls panel hierarchy, and what components it can be used to create.

Table 12–1 Data Controls Panel Icons and Object Hierarchy for Web Services

Icon	Name	Description	Used to Create...
	Data Control	<p>Represents a data control. You cannot use the data control itself to create UI components, but you can use any of the child objects listed under it. Depending on how your web services are defined, there may be more than one data control.</p> <p>Typically, there is one data control for each web service. However, you may have additional data controls that were created for other types of business services (for example, application modules). For information about creating data controls for application modules, see Chapter 11, "Using Oracle ADF Model in a Fusion Web Application".</p>	Serves as a container for other objects and is not used to create anything
	Collection	<p>Represents a named data collection. A <i>data collection</i> represents a set of data objects (also known as a <i>rowset</i>) in the data model. Each object in a data collection represents a specific structured data item (also known as a <i>row</i>) in the data model. Throughout this guide, <i>data collection</i> and <i>collection</i> are used interchangeably.</p> <p>For more information about using collections on a data control to create forms, see Chapter 20, "Creating a Basic Databound Page".</p> <p>For more information about using collections to create tables, see Chapter 21, "Creating ADF Databound Tables".</p> <p>For more information about using master-detail relationships to create UI components, see Chapter 22, "Displaying Master-Detail Data".</p> <p>For information about creating graphs, charts, and other visualization UI components, see Chapter 24, "Creating Databound ADF Data Visualization Components".</p>	Forms, tables, graphs, trees, range navigation components, and master-detail components.
	Attribute	<p>Represents a discrete data element in an object (for example, an attribute in a row). Attributes appear as children under the collections or method returns to which they belong.</p> <p>For information about using attributes to create fields on a page, see Section 20.2, "Using Attributes to Create Text Fields".</p> <p>For information about creating lists, see Chapter 23, "Creating Databound Selection Lists and Shuttles".</p>	Label, text field, date, list of values, and selection list components.
	Structured Attribute	<p>Represents a returned object that is not one of the Java primitive types (which are represented as attributes) and is also not a collection of any type. An example of a structured attribute would be a domain, which is a developer-created data type used to simplify application maintenance.</p> <p>For more information about domains, see Section 34.1, "Creating Custom, Validated Data Types Using Domains".</p>	Label, text field, date, list of values, and selection list components
	Method	<p>Represents an operation in the data control or one of its exposed structures that may accept parameters, perform some business logic and optionally return single value, a structure or a collection of those.</p> <p>For more information about using methods that accept parameters, see Section 26.2.2.1, "Using Parameters in a Method".</p>	<p>Command components</p> <p>For methods that accept parameters: command components and parameterized forms</p>

Table 12–1 (Cont.) Data Controls Panel Icons and Object Hierarchy for Web Services

Icon	Name	Description	Used to Create...
	Method Return	<p>Represents an object that is returned by a custom method. The returned object can be a single value or a collection.</p> <p>If a custom method returns anything at all, it is usually a single scalar value. However, some custom methods can return collections.</p> <p>A method return appears as a child under the method that returns it. The objects that appear as children under a method return can be attributes of the collection, other methods that perform actions related to the parent collection, and operations that can be performed on the parent collection.</p> <p>When a single-value method return is dropped, the method is not invoked automatically by the framework. You either need to also create an invoke action as an executable, or drop the corresponding method as a button to invoke the method. For more information about executables, see Section 11.6.2.2, "Executable Binding Objects".</p>	The same components as for collections and attributes.
	Operation	<p>Represents a built-in data control operation that performs actions on the parent object. Data control operations are located in an Operations node under collections or method returns, and also under the root data control node. The operations that are children of a particular collection or method return operate on those objects only, while operations under the data control node operate on all the objects in the data control.</p> <p>If an operation requires one or more parameters, they are listed in a Parameters node under the operation.</p> <p>The standard operations supported by the web service data control are for form navigation: First, Last, Next, and Previous. Because the web service data control is not an updatable data control, you cannot use built-in operations like commit, rollback, and execute.</p>	UI command components, such as buttons, links, and menus. For more information, see Section 20.4, "Incorporating Range Navigation into Forms" and Section 20.5, "Creating a Form to Edit an Existing Record" .
	Parameter	<p>Represents a parameter value that is declared by the method or operation under which it appears. Parameters appear in the Parameters node under a method or operation.</p> <p>Array and structured parameters are exposed as updateable structured attributes and collections under the data control, which can be dropped as an ADF form or an updateable table on the UI. You can use the UI to build a parameter that is an array or a complex object (not a standard Java type).</p>	Label, text, and selection list components.

12.4 Securing Web Service Data Controls

Web services allow applications to exchange data and information through defined application programming interfaces. SSL (Secure Sockets Layer) provides secure data transfer over unreliable networks, but SSL only works point to point. Once the data reaches the other end, the SSL security is removed and the data becomes accessible in its raw format. A complex web service transaction can have data in multiple messages being sent to different systems, and SSL cannot provide the end-to-end security that would keep the data invulnerable to eavesdropping.

Any form of security for web services has to address the following issues:

- The authenticity and integrity of data
- Data privacy and confidentiality
- Authentication and authorization

- Non-repudiation
- Denial of service attacks

Throughout this section the "client" is the web service data control, which sends SOAP messages to a deployed web service. The deployed web service may be:

- A web service deployed on Integrated WLS for testing purposes
- A web service running on Oracle Application Server
- A web service running anywhere in the world that is accessible through the Internet

12.4.1 WS-Security Specification

The WS-Security specification unifies multiple security technologies to make secure web services interoperable between systems and platforms. You can view the specification at

<http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-soap-message-security-1.0.pdf>.

WS-Security addresses the following aspects of web services security issues:

- Authentication and authorization
 - The identity of the sender of the data is verified, and the security system ensures that the sender has privileges to perform the data transaction.
 - The type of authentication can be a basic username/password pair transmitted in plain text, or trusted X509 certificate chains. SAML assertion tokens can also be used to allow the client to authenticate against the service, or allow it to participate in a federated SSO environment, where authenticated details are shared between domains in a vendor-independent manner.
- Data authenticity, integrity, and non-repudiation
 - XML digital signatures, which use industry-standard messages, digest algorithms to digitally sign the SOAP message.
- Data privacy
 - XML encryption that uses industry-standard encryption algorithms to encrypt the message.
- Denial of service attacks
 - Defines XML structures to time-stamp the SOAP message. The server uses the time stamp to invalidate the SOAP message after a defined interval.

12.4.2 How to Create and Use Key Stores

A web service data control can be configured for message level security using either Java Key Store (JKS) or the Oracle Wallet. For information on setting up and using Oracle Wallet, see the Oracle Technology Network at
<http://www.oracle.com/technology>.

The sections that follow describe the tasks involved in creating and using key stores. This is illustrated by creating two key stores, one to be configured on the server side, and the other on the client side (the data control side).

To create and use key stores:

1. Creating a key store using the J2SE 1.5 Keytool utility

2. Building key store private-public key pairs, which are used for encryption and signing
3. Obtaining a Certificate to issue digital signatures from a root certificating authority
4. Importing the Certificate into the key store
5. Exporting the Certificate with the public key for encryption

Note: The steps outlined in the sections that follow for requesting digital certificates are for test purposes only. Deployments intending to use web service data controls with digital signatures enabled must ensure that trusted certificates are generated compliant to the security policies of the deployment environment.

12.4.2.1 Creating a Key Store

To create a private-public key pair that can be used by the client for encryption and signing, open a command prompt and enter the command shown in [Example 12-3](#).

Example 12-3 Command to Create a Key Store

```
keytool -genkey -alias clientenckey -keyalg RSA -sigalg SHA1withRSA -keystore client.jks
```

The keytool utility prompts for key store password. Enter the key store password as welcome.

```
Enter keystore password: welcome
```

Answer the various questions prompted by the utility. The keytool utility then prompts for the key password. Enter the key password as clientenckey.

```
Enter key password for <clientenckey>:  
<RETURN if same as keystore password>: clientenckey
```

The keytool utility asks questions to determine the distinguished name (DN), which is a unique identifier and consists of the following components:

- CN= common name. This must be a single name without spaces or special characters.
- OU=organizational unit
- O=organization name
- L=locality name
- S=state name
- C=country, a two letter country code

When you accept the values at the end, a key store file `client.jks` is created in the current directory. It contains a single key pair with the alias `clientenckey` which can be used to encrypt the SOAP requests from the data control.

Next, create a key pair for digitally signing the SOAP requests made by the data control. At the command prompt run the command again, but use `clientsignkey` for the alias of the signing key pair.

Repeat the commands to create a key store for the server side, and use `serverenckey` for the encryption key pair, and `serversignkey` for the signing key pair.

To list the key entries in the key store, open a command prompt and enter the command shown in [Example 12–4](#).

Example 12–4 Command to List Key Pairs in the Key Store

```
keytool -list -keystore client.jks
```

The keytool utility prompts for the store password.

```
Enter keystore password: welcome
```

Enter the password that was used to create the key store (for example, welcome).

12.4.2.2 Requesting a Certificate

The keytool utility, by default, generates a self-signed certificate, which is a certificate whose issuer is the same as the generator of the key.

If your public key is to be distributed to the outside world to allow verification of the digital signatures you have issued, a trusted Certificate Authority (CA) must issue a certificate vouching your identity on your public key. To do this, create a Certificate request file for the signature key pair you have created and submit the request file to a CA.

At the command prompt, enter the command shown in [Example 12–5](#).

Example 12–5 Command to Create a Certificate Request File

```
keytool -certreq -file clientsign.csr -alias clientsignkey -keystore client.jks
```

The keytool utility prompts for the store and key passwords.

```
Enter keystore password: welcome
```

```
Enter key password for <clientsignkey>: clientsignkey
```

This generates a certificate request in a file called `clientsign.csr` for the public key aliased by `clientsignkey`.

When you are developing your application, you can use a CA such as Verisign to request trial certificates. Go to www.verisign.com, and navigate to Free SSL Trial Certificate and create a request. You must enter the same DN information you used when you created the key store. Verisign's certificate generation tool will ask you to paste the contents of the certificate request file generated by the keytool (in this case, `clientsign.csr`). Once all the information is correctly provided, the certificate will be sent to the email ID you have provided, and you have to import it into the key store.

Open the contents of the certificate in a text editor, and save the file as `clientsign.cer`.

You also have to import the root certificate issued by Verisign into the key store. The root certificate is needed to complete the certificate chain up to the issuer.

The root certificate vouches the identity of the issuer. Follow the instructions in the email you received from Verisign to access the root certificate, and paste the contents of the root certificate into a text file called `root.cer`.

Once you have the `root.cer` and `clientsign.cer` files created, enter the command shown in [Example 12–6](#) to import the certificates into your key store.

Example 12–6 Importing the Root Certificate

```
keytool -import -file root.cer -keystore client.jks
```

The keytool utility prompts for the store password.

```
Enter keystore password: welcome
```

Enter the password that was used to create the key store (in this case, welcome).

You can then proceed to import your public key certificate, as shown in [Example 12–7](#).

Example 12–7 Importing the Public Key Certificate

```
keytool -import -file clientsign.cer -alias clientsignkey -keystore client.jks
```

The keytool utility prompts for the store and key password.

```
Enter keystore password: welcome
```

```
Enter key password for <clientsignkey>: clientsignkey
```

Enter the appropriate passwords.

Execute the same commands to set up the trusted certificate chain in the server key store.

After the certificate chains are set up, the client and sever are ready to issue digitally signed SOAP requests.

Note: Trusted certificates are mandatory when issuing digital signatures on the SOAP message. You cannot issue digital signatures with self-signed/untrusted certificates in your key store.

12.4.2.3 Exporting a Public Key Certificate

The server must export its public key to the client so that the client can encrypt the data it sends to the server. The server can then use its corresponding private key to decrypt the data. The server's public key certificate is imported into the client key store.

At the command prompt, enter the command shown in [Example 12–8](#).

Example 12–8 Command to Export the Server's Public Key Certificate

```
keytool -export -file serverencpublic.cer -alias serverenckey -keystore server.jks
```

The keytool utility prompts for the store password.

```
Enter keystore password: welcome
```

Enter the password that was used to create the key store (in this case, welcome).

In this example, serverencpublic.cer contains the public key certificate of the server's encryption key. To import this certificate in the client's key store, enter the command shown in [Example 12–9](#).

Example 12–9 Command to Import the Server's Public Key Certificate

```
keytool -import -file serverencpublic.cer -alias serverencpublic -keystore client.jks
```

The keytool prompts for the store password.

```
Enter keystore password: welcome
```

Enter the password that was used to create the key store (in this case, welcome).

Similarly, you must export the client's encryption key so that you can import it into the server's key store. At the command prompt, enter the command shown in [Example 12–10](#).

Example 12–10 Command to Export the Client's Encryption Key

```
keytool -export -file clientencpublic.cer -alias clientenckey -keystore client.jks
```

The keytool utility prompts for the store password.

```
Enter keystore password: welcome
```

Enter the password that was used to create the key store (in this case, welcome).

Then you can import the certificate into the server's key store. Enter the command shown in [Example 12–11](#).

Example 12–11 Command to Import the Client's Public Key Certificate

```
keytool -import -file clientencpublic.cer -alias clientencpublic -keystore server.jks
```

The keytool utility prompts for the store password.

```
Enter keystore password: welcome
```

Enter the password that was used to create the key store (in this case, welcome).

The server and client key stores are now ready to be used to configure security for the web service data control.

12.4.3 How to Define Web Service Data Control Security

Once you have a web services data control in a JDeveloper project, you can define security using the Data Control Security wizard.

To define security for a web service data control:

1. In the Application Navigator, select the web service data control.
2. In the Structure window, right-click the web service data control, and choose **Define Web Service Security**.
3. On the Authentication page of the Data Control Security wizard, specify how you want to implement authentication for securing the web service. For more information, see [Section 12.4.3.1, "Setting Authentication"](#).
4. On the Message Integrity page, specify how the outbound message bodies are to be signed and how the signature on inbound message bodies is to be verified. For more information, see [Section 12.4.3.2, "Setting Digital Signatures"](#).
5. On the Message Confidentiality page, specify whether the outbound and inbound SOAP message bodies are to be encrypted and choose the encryption method. For more information, see [Section 12.4.3.3, "Setting Encryption and Decryption"](#).
6. On the Configure Key Store page, specify the location of the key store which contains the various keys used for signing, encryption, and decryption of the messages sent and received by the data control. For more information, see [Section 12.4.3.4, "Using a Key Store"](#).

-
7. On the Finish page, review your selections and click **Finish**.

12.4.3.1 Setting Authentication

WS-Security allows for service level authentication by using either username tokens or binary tokens. In addition to these, the web service client can issue SAML assertion tokens that can be used for server side authentication, or for participation in a federated SSO environment.

How authentication using a certificate is done depends on the implementation and integration with the platform security system.

- For Username Token authentication, the username/password pair must be a trusted user entry in the default security realm for the server domain on which the web service is deployed.
- For X509 Token authentication, the CN (Common Name) on whom the Certificate is issued must be a trusted user.
- For SAML authentication, the user must be a trusted user.

Note: When the application is deployed to Oracle Application Server, the administrator should use the security editing tool to add users to the security system, grouping them in the appropriate role and granting appropriate privileges.

12.4.3.1.1 Username Tokens

Username tokens provide basic authentication of a username/password pair. The passwords can be transmitted as plain text or digest.

Note: This is not the same as HTTP basic or digest authentication. The concept is similar, but it differs in that the recipient of HTTP authentication is the HTTP server, whereas for the web service data control, the username tokens are passed with the message, and the recipient is the target web service.

To use username tokens for authentication:

1. On the Authentication page of the wizard, under **Available Operations**, select one or more ports or operations to apply the authentication to.
2. Select **Username Token** as the authentication type.
3. Enter the remaining information required for username authentication.

12.4.3.1.2 X509 Certificate Authentication

An X509 certificate issued by a trusted CA is a binary security token which can be used to authenticate the client. The client sends its X509 certificate with a digital signature, which is used by the server for authentication. The X509 certificate chain associated with signature key is used for authentication.

You must have the key store file, with the root certificate of the CA, installed on the server.

Note: An X509 certificate can only be configured at port level, unlike the other authentication types that can be configured at port or operation level.

To use X509 certificate authentication:

1. On the Authentication page of the wizard, under **Available Operations**, select one or more ports to apply the authentication to.
2. Select **X509 Token** as the authentication type.
3. Later, on the Configure Key Store page of the wizard, you will need to specify the location of the key store file, and enter the signature key alias and password.

12.4.3.1.3 SAML Assertion Tokens

SAML assertion tokens can be used to allow the client to authenticate against the web service, or allow the client to participate in a federated SSO environment, where authenticated details can be shared between domains in a vendor-independent manner.

Note: SAML assertions will not be issued if the user identity cannot be established by JAZN.

To use SAML authentication:

1. On the Authentication page of the wizard, select **SAML Token** as the authentication type.
2. The **Subject Name** is the user name against which the SAML assertions will be issued.
3. From the **Confirmation** dropdown list you can select one of the following options:
 - **SENDER-VOUCHES** (default). The SAML tokens must be digitally signed. This is the preferred method to issue SAML tokens. If you choose this confirmation technique, then you must configure a key store and enter key store and signature key information on the Configure Key Store page of the wizard.
 - **SENDER-VOUCHES-UNSIGNED**. The SAML tokens are transmitted without any digital signatures. If you choose this confirmation technique, then you need not configure a key store and signature key.

12.4.3.2 Setting Digital Signatures

You can configure digital signatures on the outgoing SOAP messages, and verify digital signatures on the incoming message from the web service your application is contacting. You can also enforce an expiration window for the digital signatures.

You can set a digital signature on the outgoing SOAP message at port or operation level on the Message Integrity page of the wizard, and verify the digital signatures from the incoming message of the web service.

To sign the SOAP request, and verify the signature of the SOAP response:

1. On the Message Integrity page of the wizard, select the appropriate options.
2. Later, on the Configure Key Store page of the wizard, you will need to specify the location of the key store file, and enter the signature key alias and password.

12.4.3.3 Setting Encryption and Decryption

When you create a web service in JDeveloper, you can set security options in the Web Service Editor dialog. These are then applied at the server side when the web service is deployed. Refer to the JDeveloper online help for complete information.

Before deploying the web service, run the editor and configure encryption and decryption details on the web service. Ensure that you have specified the client's (that is, the data control's) public key to be used for encryption.

You can encrypt and outgoing SOAP message at port or operation level on the Message Confidentiality page of the wizard, and decrypt the incoming message from the web service.

To encrypt the SOAP request, and decrypt the SOAP response:

1. On the Message Confidentiality page of the wizard, select **Encrypt the SOAP Request** and select the appropriate algorithm. The encryption algorithm you select must be the same as that configured on the server side when the web service was deployed.
2. Enter the server's public key alias to allow the data control to encrypt the key details using the server's public key. For example, the alias of the server key store certificate imported in [Example 12-9](#) is `serverencpublic`.
3. If the web service uses outgoing message encryption, select **Decrypt Incoming SOAP Response**.
4. Later, on the Configure Key Store page of the wizard, you will need to specify the location of the key store file, and enter the encryption key alias and password.

12.4.3.4 Using a Key Store

[Section 12.4.2.1, "Creating a Key Store"](#) described setting up key stores for the client (the web service data control) and for the server (a deployed web service). On the Configure Key Store page of the Data Control Security wizard you enter the information needed for the key store to be used for data control security.

The final stage of configuring security for a data control based on a web service is to specify the key store details. Enter the information to access the client key store here, and when the wizard is finished the keys configured in the store will be available for signatures and encryption for all requests generated by the data control and all responses processed by the data control.

Part III

Creating ADF Task Flows

Part III contains the following chapters:

- [Chapter 13, "Getting Started with ADF Task Flows"](#)
- [Chapter 14, "Working with Task Flow Activities"](#)
- [Chapter 15, "Using Parameters in Task Flows"](#)
- [Chapter 16, "Using ADF Task Flows as Regions"](#)
- [Chapter 17, "Creating Complex Task Flows"](#)

13

Getting Started with ADF Task Flows

This chapter describes how to create ADF task flows that enable navigation, encapsulation, reuse, managed bean lifecycles, and transactions within an application. It includes the basic steps for creating a task flow diagram, adding activities and control flows to it, and running the finished task flow.

This chapter includes the following sections:

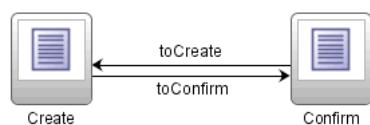
- [Section 13.1, "Introduction to ADF Task Flows"](#)
- [Section 13.2, "Creating Task Flows"](#)
- [Section 13.3, "Adding a Bounded Task Flow to a Page"](#)
- [Section 13.4, "Designating a Default Activity in an ADF Bounded Task Flow"](#)
- [Section 13.5, "Running ADF Task Flows"](#)
- [Section 13.6, "Setting Project Properties for ADF Task Flows"](#)
- [Section 13.7, "Refactoring to Create New ADF Task Flows and Templates"](#)
- [Section 13.8, "What You Should Know About Task Flow Constraints"](#)

13.1 Introduction to ADF Task Flows

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. Each task flow contains a portion of the application's navigational graph. The nodes in the task flows are activities. An *activity node* represents a simple logical operation such as displaying a view, executing application logic, or calling another task flow. The transactions between the activities are called *control flow cases*.

As shown in [Figure 13–1](#), an activity is represented as a node in an ADF task flow diagram. [Figure 13–1](#) contains two view activities called `Create` and `Confirm`. These view activities are similar to page nodes within a JSF page flow.

Figure 13–1 ADF Task Flow



13.1.1 Task Flow Advantages

ADF task flows offer significant advantages over standard JSF page flows, as described in [Table 13–1](#).

Table 13–1 ADF Task Flow Advantages

JSF Page Flow	ADF Task Flow
The entire application must be represented within a single JSF page flow.	The application can be broken up into a series of task flows that call one another.
All nodes within a JSF page flow must be JSF pages. No other types of objects can exist within the JSF page flow.	You can add to the task flow diagram nodes such as views, method calls, and calls to other task flows.
Navigation is only between pages.	Navigation is between pages as well as other activities, including routers (see Section 14.4, "Using Router Activities").
Application fragments cannot be reused.	ADF task flows are reusable within the same or an entirely different application.
There is no shared memory scope between multiple requests except for session scope.	After you break up your application into task flows, you may decide to reuse task flows containing common functionality.
	Shared memory scope (for example, page flow scope) enables data to be passed between activities within the task flow. Page flow scope defines a unique storage area for each instance of an ADF bounded task flow

13.1.2 Task Flow Types

The two types of ADF task flow are:

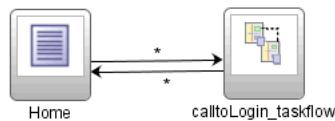
- Unbounded task flow: A set of activities, control flow rules and managed beans that interact to allow a user to complete a task. An ADF unbounded task flow consists of all activities and control flows in an application that are not included within any bounded task flow.
- Bounded task flow: A specialized form of task flow that 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. An ADF bounded task flow allows reuse, parameters, transaction management, and reentry.

[Table 14–1](#) describes the activity types that you can add to an ADF unbounded or bounded task flow.

A bounded task flow is also called a *task flow definition*. A task flow definition source file contains the metadata for the bounded task flow. Multiple task flow definitions can be included within the same task flow definition file.

A typical application is a combination of an unbounded and one or more bounded task flows. Every Fusion web application contains an unbounded task flow, even if the unbounded task flow is empty. The application can then call bounded task flows from activities within the unbounded task flow.

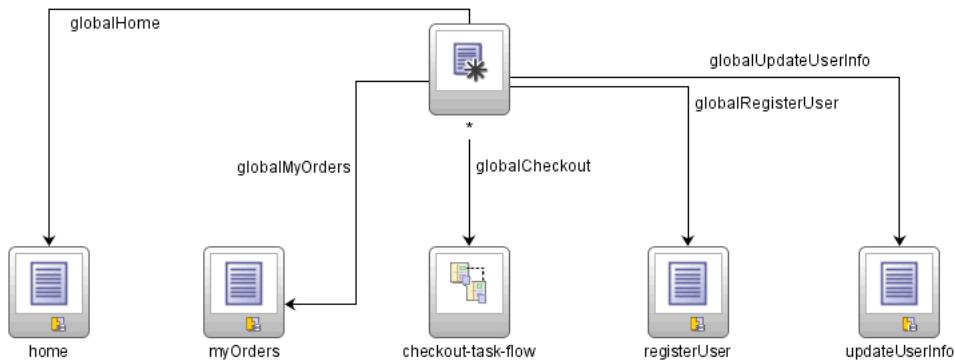
As shown in [Figure 13–2](#), the first activity to execute in an application is often a view activity within an ADF unbounded task flow. A view activity represents a JSF page that displays as part of the application. The activity shown in [Figure 13–2](#) starts with the Home view activity and then calls a bounded task flow. The `callToLogin_taskFlow` activity calls a bounded task flow that enables a user to log into the application.

Figure 13–2 Unbounded Task Flow Calling a Bounded Task Flow

You can also design an application in which all application activities reside within the ADF unbounded flow. This mimics a Struts or JSF application, but doesn't take advantage of ADF bounded task flow functionality. To fully take advantage of task flow functionality, use ADF bounded task flows.

13.1.2.1 ADF Unbounded Task Flows

A Fusion web application always contains an *ADF unbounded task flow*, which contains the entry point or points to the application. Figure 13–3 displays the diagram for the unbounded task flow from the Fusion Order Demonstration application. This task flow contains the `Home`, `MyOrders`, `registerUser`, and `updateUserInfo` view activities, which are all entry points to the application.

Figure 13–3 Unbounded Task Flow in Fusion Order Demonstration Application

You typically use an unbounded instead of a bounded task flow if:

- You want to take advantage of ADF Controller features not offered by bounded task flows, such as bookmarking view activities. For more information, see [Section 14.2.3, "Bookmarking View Activities"](#).
- You don't need ADF Controller features that are offered when using a bounded task flow.
- The task flow will not be called by another task flow.
- The application has multiple points of entry. In Figure 13–3, the task flow can be entered through any of the pages represented by the view activity icons on the unbounded task flows.

Pages are associated with view activities. The icon for a view activity displays a page image like this.



- You want to bookmark more than one activity on the task flow. See [Section 14.2.3, "Bookmarking View Activities"](#) for more information.

An unbounded task flow cannot declaratively specify parameters. In addition, it cannot contain a *default activity*, an activity designated as the first to run in the unbounded task flow. This is because an unbounded task flow does not have a single point of entry. To perform any of these requires an ADF bounded task flow.

In order to take advantage of completely declarative ADF Controller transaction and reentry support, use a bounded rather than an unbounded task flow.

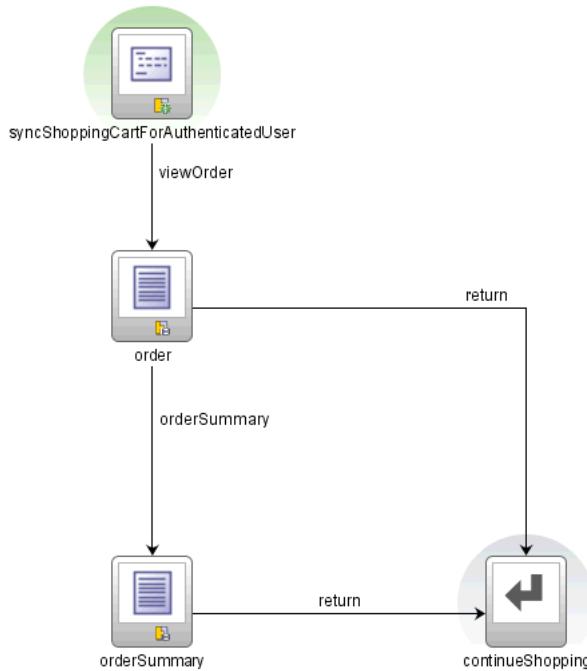
13.1.2.2 ADF Bounded Task Flows

An *ADF bounded task flow* is used to encapsulate a reusable portion of an application. A bounded task flow is similar to a Java method in that it:

- Has a single entry point
- May accept input parameters
- May generate return values
- Has its own collection of activities and control flow rules
- Has its own memory scope and managed bean lifespan (a page flow scope instance)

The `checkout-task-flow` activity in [Figure 13–3](#) is a call to an ADF bounded task flow. An unbounded task flow can call an ADF bounded task flow, but cannot be called by another task flow. A bounded task flow can call another bounded task flow, which can call another and so on. There is no limit to the depth of the calls.

The checkout process is created as a separate ADF bounded task flow, as shown in [Figure 13–4](#).

Figure 13–4 Checkout Bounded Task Flow in Fusion Order Demonstration Application

The reasons for creating the `checkout-task-flow` activity as a called bounded task flow are:

- The bounded task flow always specifies a default activity, a single point of entry that must execute immediately upon entry of the bounded task flow.

In the checkout task flow, the activity labeled

`syncShoppingCartForAuthenticatedUser` is a call to a method that returns a list of items that an anonymous user (one who has not yet logged in to the application) may have chosen to purchase. Any items chosen before authentication are included in the shopping cart after the user has logged in. Because it is the default activity, the method is always invoked before the shopping cart order page displays.

- `checkout-task-flow` is reusable. For example, it can be included in other applications requiring an item checkout process. The bounded task flow can also be reused within the same application.
- Any managed beans you decide to use within `checkout-task-flow` can be specified in page flow scope, so are isolated from the rest of the application.

The main features of ADF bounded task flows are summarized in [Table 13–2](#).

Table 13–2 ADF Bounded Task Flow Features

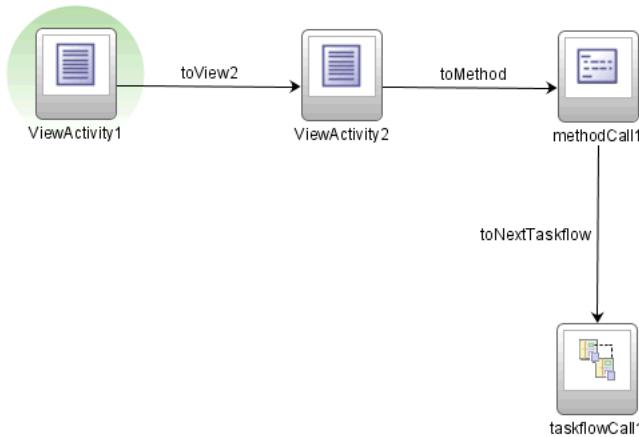
Feature	Description
Well-defined boundary	An ADF bounded task flow consists of its own set of private control flow rules, activities, and managed beans. A caller requires no internal knowledge of such things as page names, method calls, child bounded task flows, managed beans, and control flow rules within the bounded task flow boundary. Input parameters can be passed into the bounded task flow, and output parameters can be passed out on exit of the bounded task flow. Data controls can be shared between task flows.
Single point of entry	An ADF bounded task flow has a single point of entry, a default activity that executes before all other activities in the task flow. For more information, see Section 13.4, "Designating a Default Activity in an ADF Bounded Task Flow" .
pageFlow memory scope	You can specify pageFlow as the memory scope for passing data between activities within the ADF bounded task flow. pageFlow scope defines a unique storage area for each instance of an ADF bounded task flow. Its lifespan is the ADF bounded task flow, which is longer than request scope and shorter than session scope. For more information, see Section 13.2.10, "What You May Need to Know About Memory Scopes" .
Addressable	You can access an ADF bounded task flow by specifying its unique identifier within the XML source file for the bounded task flow and the file name of the XML source file. For more information, see Section 14.6.7, "What Happens When You Add a Task Flow Call Activity" .
Reuse	You can identify an entire group of activities as a single entity, an ADF bounded task flow, and reuse the bounded task flow in another application within an ADF region. For example, the Hot Items and Start Shopping tabs on the home page of the Fusion Order Demonstration application reuse the same task flow embedded in a region. Different parameters are passed to each region to determine the lists of products that display. For more information, see Section 16.1, "Introduction to ADF Regions" . You can also reuse an existing bounded task flow simply by calling it. For example, one task flow can call another bounded task flow using a task flow call activity or a URL. In addition, you can use task flow templates to capture common behaviors for reuse across different ADF bounded task flows. For more information, see Section 17.8, "Creating an ADF Task Flow Template" .
Parameters and return values	A caller can pass input parameters to an ADF bounded task flow and accept return values from it. For more information, see Section 15.2, "Passing Parameters to an ADF Bounded Task Flow" . In addition, you can share data controls between bounded task flows. For more information, see Section 15.3, "Sharing Data Control Instances" .
Transaction management	An ADF bounded task flow can represent a transactional unit of work. You can declaratively specify options on the task flow definition that determine whether, when entering the task flow, the task flow creates a new transaction, joins an existing one or is not part of the existing transaction. For more information, see Section 17.2, "Managing Transactions" .
Reentry	You can specify options on the task flow definition that determine whether or not the ADF bounded task flow can be reentered. For more information, see Section 17.3, "Reentering an ADF Bounded Task Flow" .

Table 13–2 (Cont.) ADF Bounded Task Flow Features

Feature	Description
On-demand loading of metadata	ADF bounded task flow metadata is loaded on demand when entering an ADF bounded task flow.
Security	You can secure an ADF bounded task flow by defining the privileges that are required for someone to use it.

13.1.3 Control Flows

A task flow consists of activities and control flow cases that define the transitions between activities. In [Figure 13–5](#), the control flow labeled `toView2` defines the transition between `ViewActivity1` and `ViewActivity2`. When the task flow executes, `ViewActivity1` displays prior to `ViewActivity2`.

Figure 13–5 Task Flow with Activities and Control Flow Cases

[Figure 13–5](#) contains a method call. The method's logic is invoked after `ViewActivity2` and before it calls a bounded task flow. In a task flow, you have the option of invoking an activity such as a method before or after a page is rendered. Invoking logic outside of a particular page can facilitate reuse because the page can be reused in other contexts that don't require the method (for example, within a different task flow).

Control flow rules are based on JSF navigation rules, but capture additional information. JSF navigation is always between pages, whereas control flow rules describe transitions between activities. For example, a control flow rule can indicate a transition between a view activity and a subsequent method call activity. Or, it can indicate that control passes from the page to another task flow.

Save point restore, task flow return, and URL view activities cannot be the source of a control flow rule.

The basic structure of a control flow rule mimics a JSF navigation rule. [Table 13–3](#) describes how metadata maps from JSF navigation rules to control flow rules.

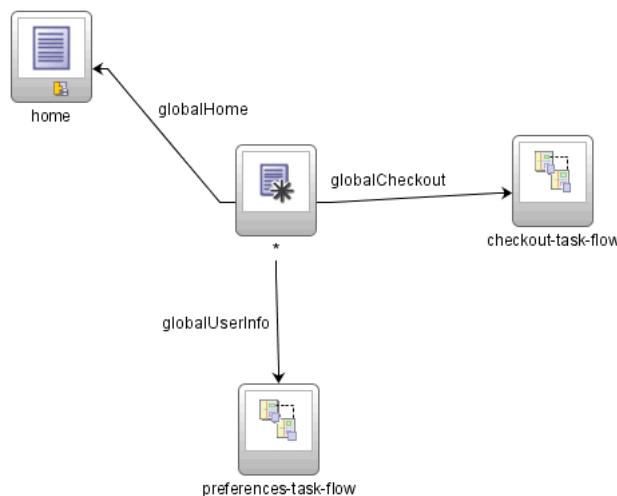
Table 13–3 Mapping of JSF Navigation Rules to Control Flow Rules

JSF Navigation Rule	Control Flow Rule
Navigation Rule	Control Flow Rule

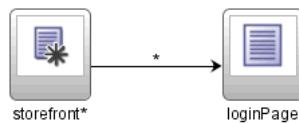
Table 13–3 (Cont.) Mapping of JSF Navigation Rules to Control Flow Rules

JSF Navigation Rule	Control Flow Rule
From View ID	From Activity ID
Navigation Case	Control Flow Case
From Action	From Action
From Outcome	From Outcome
To View ID	To Activity ID

A wildcard control flow rule represents a control flow `from-activity-id` that contains a trailing wildcard (`foo*`) or a single wildcard character (*). In [Figure 13–6](#), the wildcard control flow rule contains a single wildcard character, indicating that control can pass to the activities connected to it in the task flow diagram from any activity within the task flow.

Figure 13–6 Wildcard Control Flow Rule

The trailing wildcard in [Figure 13–7](#) indicates that control flow can pass to the `loginPage` view from any valid source activity whose `activity-id` begins with the characters `storefront`.

Figure 13–7 Wildcard Control Flow Rule with Trailing Wildcard

13.2 Creating Task Flows

The processes for creating ADF bounded and unbounded task flows are similar. The main difference is that you select the **Create as Bounded Task Flow** checkbox in the

Create ADF Task Flow dialog (shown in [Figure 13–8](#)) to create an ADF bounded task flow.

Before you create a task flow, you must already have an application and project (see [Chapter 1, "Introduction to Building Fusion Web Applications with Oracle ADF"](#) for more information). You'll use the project to store any JSF pages, Java code, and XML source files associated with the task flow.

Note: When you create the project, you may not need to create an unbounded task flow for it. If **ADF Page Flow** is specified as a Selected Technology on the Technology Scope page of the Project Properties dialog, the new `adfc-config.xml` source file is automatically created within the project. `adfc-config.xml` is the main source file for an unbounded task flow.

13.2.1 How to Create an ADF Task Flow

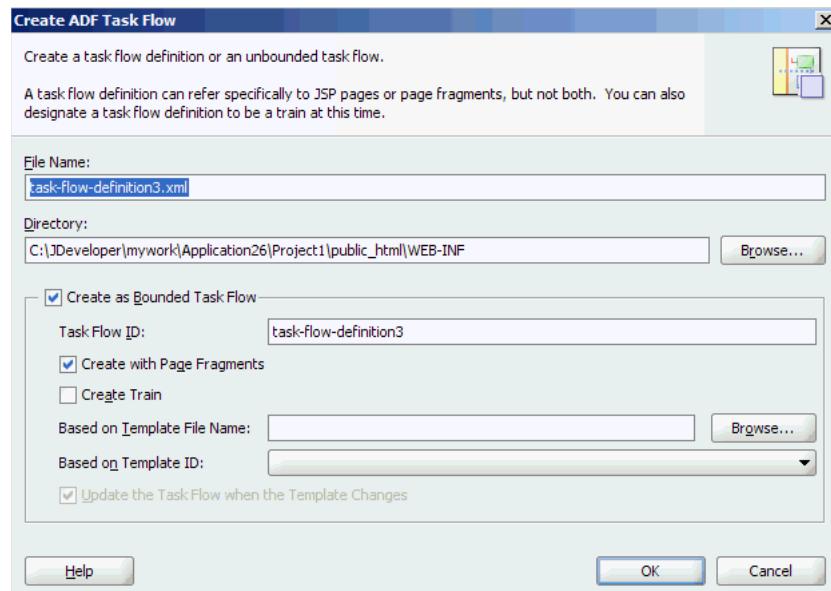
The steps for creating an ADF unbounded task flow or an ADF bounded task flow are described below.

To create an ADF unbounded or bounded task flow:

1. In the Application Navigator, right-click the project where you want to create the task flow and choose **New**.
2. In the New Gallery, expand **Web Tier** node, select **JSF** and then **ADF Task Flow** and click **OK**.

The dialog shown in [Figure 13–8](#) displays.

Figure 13–8 Create ADF Task Flow Dialog



3. In the Create ADF Task Flow dialog, the **Create as Bounded Task Flow** checkbox is selected by default. Deselect it to create a source file that will be incorporated into the application's unbounded task flow.

Deselecting the checkbox automatically changes the default value in the **File Name** field. This value will be used to name the XML source file for the ADF task flow you are creating. The XML source file contains metadata describing the activities and control flow rules in the task flow:

- The default name for an unbounded task flow is `adfc-config.xml`.
- The default source file name for a bounded task flow matches the value specified in the **Task Flow ID** field.

Because a single project can contain multiple task flows, a number may be added to the default value in the **File Name** field in order to give the source file a unique name, for example, `task-flow-definition3.xml`.

4. Click **OK**.

A diagram representing the task flow displays in the editor.

Tip: You can view a thumbnail of the entire task flow diagram by clicking the diagram and then choosing **View > Thumbnail** from the main menu.

5. After you create the task flow, you can update it in the task flow editor using the:

- **Diagram** tab: A visual representation of a single task flow. Each node in the task flow diagram corresponds to an activity. Each line connecting activities corresponds to a control flow case.
- **Source** tab: The XML source file for a task flow.
- **Overview** tab: Options corresponding to metadata elements in the source file.

You can also use the Structure window to update the task flow.

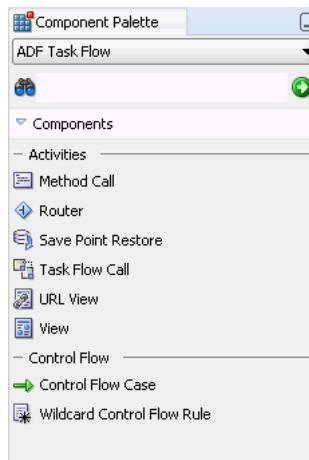
Tip: There are other ways to create task flows, for example, by refactoring contents of an existing ADF task flow into a new task flow. For more information, see [Section 13.7, "Refactoring to Create New ADF Task Flows and Templates"](#).

13.2.2 How to Add an Activity to an ADF Task Flow

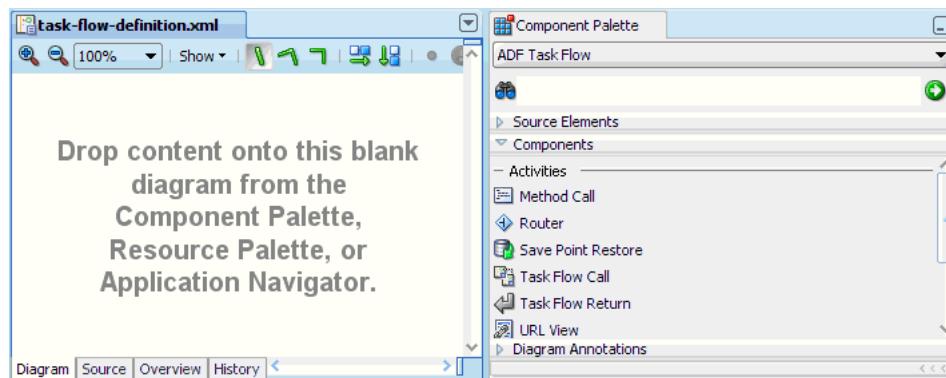
After you create a task flow, a task flow diagram and the Component Palette automatically display. The task flow diagram is a visual editor on which you can add the activities and control flows for the task flow. You can add them to the diagram by dragging them from the Component Palette.

As shown in [Figure 13–9](#), the Component Palette contains separate sections for components and diagram annotations. The contents of the **Components** section differ slightly depending on whether you are creating an ADF bounded or an unbounded task flow. For example, if you are creating an ADF bounded task flow, the **Components** section contains an additional task flow return activity.

[Table 14–1](#) displays the activities you can add to an ADF task flow.

Figure 13–9 ADF Unbounded Task Flow Component Palette**To add an activity to an ADF task flow:**

1. In the Application Navigator, double-click a task flow source file, for example, `adf-c-config.xml`, to display the task flow diagram.
You can find XML source files under the Page Flows node in the WEB-INF node for the project containing the task flow.
2. In the task flow editor, the **Diagram** tab displays the task flow diagram, as shown in [Figure 13–10](#).

Figure 13–10 Task Flow Diagram

Notice that the Component Palette automatically displays ADF task flow components.

3. Drag an activity from the ADF Task Flow page in the Component Palette onto the diagram.
 - If you drag a view activity onto the diagram, you can double-click it to display the Create JSF JSP Page wizard, where you can define characteristics for the page or page fragment. For more information, see [Section 14.2, "Using View Activities"](#).
 - If you drag a task flow call activity onto the diagram, you can double-click it to display the Create ADF Task Flow dialog where you can define settings for

a new bounded task flow. For more information, see [Section 14.6, "Using Task Flow Call Activities"](#).

Tip: For each activity you drag to the task flow diagram, you can display optional status icons and a tooltip that provide additional information about the activity. For example, after you drag a view activity to the task flow diagram, it may display a warning icon until you associate it with a JSF page.

To turn on the icons, select **Show** at the top of the task flow diagram, and then select **Status** and one of the following:

- **Error:** Displays when there is a problem in the task flow metadata which prevents it from running. For example, a view activity in the metadata can contain a `<bookmark>` or `<redirect>` element, but not both.
- **Warning:** Displays when an activity is incomplete. For example, a view activity that doesn't have a physical page associated with it or a task flow call that doesn't have a task flow reference associated with it are both considered incomplete activities. The resulting task flow metadata may prevent it from running.

You can drag your mouse over a secondary icon to read a tool tip describing the icon's meaning.

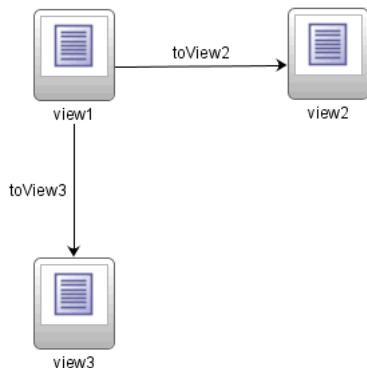
13.2.3 How to Add Control Flows

A control flow case identifies how control passes from one activity to the next in the application. To create a control flow, you select **Control Flow Case** in the ADF Task Flow page of the Component Palette and drag it from a source activity to a target activity. Dragging the control flow case between the two activities automatically creates a control flow comprised of the following:

- `control-flow-rule`: Identifies the source activity using a `from-activity-id`, for example, `view1`.
- `control-flow-case`: Identifies the target activity using a `to-activity-id`, for example, `view2`.

Once you identify an activity as the source for a control flow case, any additional control flow case that has the activity as its source is organized under the same control flow rule. In [Figure 13–11](#), there is one `control-flow-rule` that identifies `view1` as the source activity, and two `control-flow` cases that identify the target activities `view2` and `view3`.

There can be multiple control flow cases for each control flow rule. The order of control flow cases is determined by the order in which they appear in the `adfc-config.xml` file.

Figure 13–11 Multiple Control Flow Cases from a Single Activity

Example 13–1 shows the metadata for the multiple control flow cases shown in Figure 13–11.

Example 13–1 Control Flow Rule Metadata within a Task Flow Definition

```

<task-flow-definition id="task-flow-definition">
    .
    .
    .
    <control-flow-rule>
        <from-activity-id>view1</from-activity-id>
        <control-flow-case>
            <from-outcome>toView2</from-outcome>
            <to-activity-id>view2</to-activity-id>
        </control-flow-case>
        <control-flow-case>
            <from-outcome>toView3</from-outcome>
            <to-activity-id>view3</to-activity-id>
        </control-flow-case>
    </control-flow-rule>
    .
    .
    .
</task-flow-definition>
  
```

Use the task flow diagram as a starting point for creating basic control flows between activities. Later, you can edit control flow properties in the Structure window, Property Inspector or overview editor for the task flow diagram.

To define a control flow case directly in the task flow diagram:

1. In the Application Navigator, double-click a task flow source file, for example, `adfc-config.xml`, to display the task flow diagram.
2. In the ADF Task Flow page of the Component Palette, select **Control Flow Case**.
3. On the diagram, click a source activity, for example a view, and then click the destination activity, as shown in [Figure 13–12](#).

Figure 13–12 Control Flow Case

JDeveloper adds the control flow case to the diagram. Each line that JDeveloper adds between activities represents a control flow case. The arrow indicates the direction of the control flow case. JDeveloper automatically adds a default wildcard (*) from-outcome value as the control flow label. The from-outcome contains a value that can be matched against values specified in the action attribute of UI components.

4. To change the `from-outcome`, select the text next to the control flow in the diagram. By default, this text is the wildcard * character as shown in [Figure 13–12](#). You can overwrite the text with a new `from-outcome`, for example, `toView2`.
5. To change the `from-activity-id` (identifies the source activity) or `to-activity-id` (identifies the target activity), drag either end of the arrow in the diagram to a new activity.

Tips: After you select the control flow in the task flow diagram, you can also change its properties in the Property Inspector or the Structure window. The Structure window is helpful for displaying the relationship between control rules and cases.

You can also click **Control Flows** on the **Overview** tab for the task flow diagram to add cases, as shown in [Figure 13–13](#). To add a case, make sure that the **From Activity** (source activity) and the **To Activity** (target activity) for the rule have already been added to the task flow.

Figure 13–13 Control Flows on Task Flow Editor Overview Tab

Control Flows		
From Activity	From Outcome	To Activity
view1	toView2	view2
view1	toView3	view3

13.2.4 How to Add a Wildcard Control Flow Rule

You can add a wildcard control flow rule to an unbounded or bounded task flow. The steps for adding it are similar to those for adding any activity to a task flow diagram.

To add a wildcard control flow rule:

1. In the Application Navigator, double-click a task flow source file, for example, `adfc-config.xml`, to display the task flow diagram.
2. Drag **Wildcard Control Flow Rule** from the ADF Task Flow page of the Component Palette and drop it on the task flow diagram.
3. Select **Control Flow Case** from the ADF Task Flow page list of the Component Palette.

4. In the task flow diagram, drag the control flow case from the wildcard control flow rule to the target activity.

The target can be any activity type.

5. By default, the label below the wildcard control flow rule is *, which corresponds to a single * character in its `from-activity-id` value. To change this value, select the wildcard control flow rule in the diagram. In the Property Inspector for the wildcard control flow rule, enter a new value in the **from_activity_id_field**, for example, `project*`. The wildcard must be a trailing character in the new label.

Tip: You can also change the `from-activity-id` value in the Overview editor for the task flow diagram.

13.2.5 What Happens When You Create a Control Flow Rule

Understanding the elements that define the rules in the source file for the task flow helps when creating control flow rules directly in the ADF task flow diagram, ADF task flow overview editor, or Structure window, or when adding them directly in the XML source file. [Example 13–2](#) shows the general syntax of a control flow rule element in the task flow source file.

Example 13–2 Control Flow Rule Syntax in the Source File

```
<control-flow-rule>
    <from-activity-id>from-view-activity</from-activity-id>
    <control-flow-case>
        <from-action>actionmethod</from-action>
        <from-outcome>outcome</from-outcome>
        <to-activity-id>destinationActivity</to-activity-id>
    </control-flow-case>
    <control-flow-case>
        .
        .
        .
    </control_flow-case>
</control-flow-rule>
```

Control flow rules can consist of the following metadata:

- `control-flow-rule`: A mandatory wrapper element for control flow case elements.
- `from-activity-id`: The identifier of the activity where the control flow rule originates, for example, `source`.

A trailing wildcard (*) character in `from-activity-id` is supported. The rule will apply to all activities that match the wildcard pattern. For example, `login*` matches any logical activity ID name beginning with the literal `login`. If you specify a single wildcard character in the metadata (not a trailing wildcard), the control flow automatically converts to a wildcard control flow rule activity in the diagram. For more information, see [Section 13.2.4, "How to Add a Wildcard Control Flow Rule"](#).

- `control-flow-case`: A mandatory wrapper element for each case in the control flow rule. Each case defines a different control flow for the same source activity. A control flow rule must have at least one control flow case.
- `from-action`: An optional element that limits the application of the rule to outcomes from the specified action method. The action method is specified as an EL binding expression, such as `# {backing(bean).cancelButton_action}`.

In [Example 13–2](#), control passes to destinationActivity only if outcome is returned from actionmethod.

The value in from-action applies only to a control flow originating from a view activity, not from any other activity types. Wildcards are not supported in from-action.

- from-outcome: Identifies a control flow case that will be followed based on a specific originating activity outcome. All possible originating activity outcomes should be accommodated with control flow cases.

If you leave both the from-action and the from-outcome elements empty, the case applies to all outcomes not identified in any other control flow cases defined for the activity, thus creating a default case for the activity. Wildcards are not supported in from-outcome.

- to-activity-id: A mandatory element that contains the complete identifier of the activity to which the navigation is routed if the control flow case is performed. Each control flow case can specify a different to-activity-id.

13.2.6 What Happens at Runtime

At runtime, the ADF Controller evaluates control flow rules from the most specific to the least specific match to determine the next transition between activities. Evaluation is based on the following priority, which is similar to that for JSF navigation rules:

1. from-activity-id, from-action, from-outcome
2. from-activity-id, from-outcome
3. from-activity-id

ADF Controller first searches for a match in all three elements: from-activity-id, from-action, and from-outcome. If there is no match, ADF Controller searches for a match in just the from-activity-id and from-outcome elements. Finally, ADF Controller searches for a match in the from-activity-id element alone.

If ADF Controller cannot find a control flow rule within its metadata to match a request, it will allow the standard JSF navigation handler to find a match.

An unbounded task flow can have more than one ADF Controller XML source file. Because control flow rules can be defined in more than one ADF Controller XML source file, similar rules may be defined in different files. If there is a conflict in which two or more cases have the same from-activity-id and the same from-action or from-outcome values, the last case (as listed in the adfc-config.xml, bootstrap, or task flow definition source file) is used. If the conflict is among rules defined in different source files, the rule in the last source file to be loaded is used.

13.2.7 What Happens When You Create an ADF Task Flow

A new XML source file is created every time you create a new ADF unbounded or bounded task flow. By default, the XML source file for an ADF unbounded task flow is called adfc-config.xml. Another configuration file, adf-config.xml, is also created automatically (for more information, see [Section A.10, "adf-config.xml"](#) for more information).

As shown in [Example 13–3](#), <adfc-config> appears first as the top-level element in all ADF Controller XML source files. Bounded task flows, activities and control flow rules are defined inside the <adfc-config> element. Bounded task flows are identified within the source file by the <task-flow-definition> metadata element.

Example 13–3 ADF Bounded Task Flow XML Source File

```
<?xml version="1.0" encoding="windows-1252" ?>
<adfc-config xmlns="http://xmlns.oracle.com/adf/Controller">
  <task-flow-definition id=<"task-flow-definition">
    <use-page-fragments/>
  </task-flow-definition>
</adfc-config>
```

An ADF bounded task flow is identified by its task flow reference, which is comprised of a unique combination of identifier and document name. [Example 13–4](#) shows a sample task flow reference within a task flow call activity.

Example 13–4 Task Flow Reference

```
<adfc-config xmlns="http://xmlns.oracle.com/adf/Controller" version="1.2">
  <task-flow-definition id="task-flow-definition">
    <use-page-fragments/>
    .
    .
    .
    <task-flow-call id="taskFlowCall">
      <task-flow-reference>
        <document>/WEB-INF/target-task-flow-definition.xml</document>
        <id>my-task-flow</id>
      </task-flow-reference>
    </task-flow-call>
    .
    .
    .
  </task-flow-definition>
</adfc-config>
```

Note: If you use JDeveloper to create the ADF bounded task flow, specify only one id (indicating one task flow definition) per document.

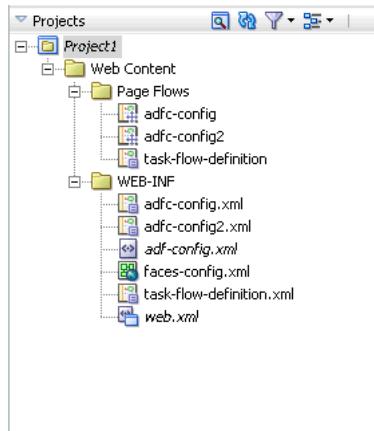
You assign both identifier and document name when you create the ADF bounded task flow. As shown in [Example 13–4](#), the identifier is the value in the **Task Flow ID** field. The document name is the value in the **File Name** field.

13.2.8 What Happens at Runtime: Using ADF Task Flows

A single application can have multiple ADF unbounded task flow XML source files and multiple ADF bounded task flow XML source files. The set of files that are combined to produce the ADF unbounded task flow is referred to as the application's *ADF Controller bootstrap configuration files*. An ADF unbounded task flow is assembled at runtime by combining one or more ADF Controller bootstrap configuration files. All activities within the bootstrap configuration files that are not contained within an ADF bounded task flow are considered to be within the ADF unbounded task flow.

For more information, see [Section 17.10, "How to Specify Bootstrap Configuration Files"](#).

The names of the source files within a single application must be different. The example in [Figure 13–14](#) contains two unbounded task flows (adfc-config, adfc-config2) and a bounded task flow (task-flow-definition).

Figure 13–14 Application with Two ADF Unbounded Task Flow XML Source Files

13.2.9 What You May Need to Know About Managed Beans

Managed beans are Java classes that you register with the application using various configuration files. For more information, see [Section 18.4, "Using a Managed Bean in a Fusion Web Application"](#). Managed beans of any scope should be defined in the `adfc-config.xml` file. The advantages of doing this are:

- All managed beans are handled consistently,
- Managed bean definitions of session, application, and request scope can reside in any task flow, not just the top-level one. That allows for better encapsulation of task flows so that they don't have to depend on things defined externally to the task flow.

All memory scopes are supported for managed beans in `adfc-config.xml`. ADF bounded task flows allow managed bean scopes of application, session, request, pageFlow, or none.

You can bind values to `pageFlow` scope managed bean properties before control is passed to a called ADF bounded task flow. `pageFlow` scoped beans can be defined in ADF Controller source files only (`adfc-config.xml` and other bootstrap source files, and task flow definition files). They are not allowed in the `faces-config.xml` file.

In order to allow managed bean customizations in an ADF bounded task flow, ensure the managed bean metadata resides in the task flow definition for the ADF bounded task flow. Avoid placing managed beans within the `faces-config.xml` file.

When an ADF bounded task flow executes, `faces-config.xml` is first checked for managed bean definitions, then the task flow definition file for the currently executing task flow is checked. If no match is found in either location and the managed bean is in session or application scope, `adfc-config.xml` and other bootstrap XML source files are checked.

Lookup precedence is based on memory scope. request scope managed beans take precedence over session scope managed beans. For example, a request scope managed bean named abc in `adfc-config.xml` takes precedence over a session scope managed bean named abc in the current file.

An instantiated bean takes precedence over a new instantiated instance. For example, an existing session scope managed bean named abc takes precedence over a request scope bean named abc defined in the current task flow definition file.

13.2.10 What You May Need to Know About Memory Scopes

Each ADF bounded and unbounded task flow exists inside a page flow scope that is independent of the memory scope for all other task flows. Page flow scope is not a single state like session scope. It defines a unique storage area for each instance of an ADF bounded task flow. Each instance of the called ADF bounded task flow has its own state. An ADF bounded task flow may therefore be called recursively.

Page flow scope is recommended as a means for passing data values between activities within a task flow. Application and session scopes are also allowed within task flow definition files, but are not recommended.

Best Practice:

Avoid using application and session scope variables because they can persist in memory beyond the life of the ADF task flow. This may compromise the encapsulation and reusable aspects of a task flow.

In addition, application and session scopes may keep things in memory longer than needed, causing overhead.

- Use the session scope only for information that is relevant to the whole session, such as user or context information. Avoid using session scope to pass values from one ADF task flow to another. Instead use page flow scope variables within the ADF task flow, and use parameters to pass values between task flows. Using parameters gives your task flow a clear contract with other task flows that call it or are called by it.
- Use view scope for variables that are needed only within the current view activity and not across view activities.
- Use request scope when the scope does not need to persist longer than the current request.
- Use backing bean scope managed beans in your task flow if there is a possibility that your task flow will appear in two region components on the same page and you would like to isolate region instances.

As a rule, use the narrowest scope that will work.

Overuse of `sessionScope` variables can be a maintenance problem. There isn't a good way to know all the variables that are being used in different places of the application. There is also the risk that other users will use the same variable and that you'll overwrite each other's settings. Using parameters gives your task flow a clear contract with other task flows that call it or are called by it. If a task flow needs a particular `sessionScope` variable to be set elsewhere, that fact might not be obvious to someone else using your task flow.

Page flow scope does not support implicit access. For example, you cannot specify `# {inpTxtBB.uiComponent}` and assume page flow scope. An EL expression that specifies page flow scope must be explicitly qualified, for example, `# {pageFlowScope.inpTxtBB.uiComponent}`.

When one task flow calls another, the calling task flow cannot access the called task flow's page flow scope and vice versa. For example, a UI component on a page in a bounded task flow cannot access the page flow scope of another task flow, even if the

task flow is an ADF region embedded in the page (for more information, see [Section 16.1, "Introduction to ADF Regions"](#)). Instead, the UI component and the ADF region must pass parameter values to one another or share data controls.

13.2.10.1 View Scope

A view scope instance exists for each view port that the ADF Controller manages, for example, a root browser window or a region. The lifespan of a view scope instance begins and end when the current `viewId` of a view port is changed. If you specify view scope, JDeveloper retains objects used on a page as long as the user continues to interact with the page. These objects are automatically released when the user leaves the page.

13.2.10.2 Backing Bean Scope

A backing bean is a managed bean that contains accessors for a UI component. Backing bean scope has a lifespan of a single request and provides an isolated instance of memory for backing beans. Other than having a different scope, management of backing beans in this scope is no different from other application beans already managed by ADF, such as page flow scope and view scope beans.

13.3 Adding a Bounded Task Flow to a Page

After you create a bounded task flow, you can drop its task flow definition from the Application Navigator onto a JSF page:

- If the bounded task flow contains view activities that are page fragments, it is added to the JSF page as an ADF region. For more information, see [Section 16.1, "Introduction to ADF Regions"](#).
The Fusion Order Demo application, for example, contains an ADF bounded task flow that displays a summary of items that the end user has added to a shopping cart. All view activities in the bounded task flow are page fragments, so the bounded task flow will be dropped on the Fusion Order Demo Application Home page as a region. The contents of the shopping cart summary will display within a physical region on the Home page.
- If the bounded task flow contains view activities that are pages, you have the option of adding a button or link to the JSF page. When clicked, the button or link executes the bounded task flow.

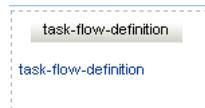
To add a bounded task flow containing pages to a JSF page:

1. In the Application Navigator, drag the ADF bounded task flow onto the page and drop it where you want the button or link to be located.

You can find task flows in the Application Navigator under the Page Flows or WEB-INF nodes.

2. Choose **Create** and either **Task Flow Call as Button** or **Task Flow Call as Link**.

As shown in [Figure 13-15](#), the task flow definition appears on the JSF page as either a button or link UI component.

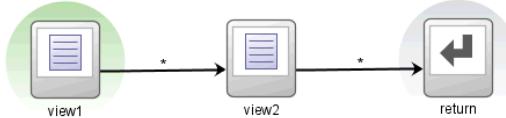
Figure 13–15 Button and Link UI Components

13.4 Designating a Default Activity in an ADF Bounded Task Flow

The default activity is the first activity to execute in an ADF bounded task flow. For example, the default activity always executes first when a task flow call activity passes control to the ADF bounded task flow.

ADF unbounded task flows do not have default activities.

As shown in [Figure 13–16](#), a green circle identifies the default activity in a task flow diagram.

Figure 13–16 Default Activity in ADF Bounded Task Flow

The first activity that you add to a new ADF bounded task flow diagram is automatically identified as the default activity. You can also right-click any activity in the task flow diagram and choose **Mark Activity > Default Activity**. The default can be any activity type and it can be located anywhere in the control flow of the ADF bounded task flow. To find the default activity, right-click anywhere on the task flow diagram and choose **Go to Default Activity**.

An ADF bounded task flow can have only one default activity. If you mark a second activity as the default, the first is unmarked automatically. To unmark an activity manually, right-click the activity in the task flow diagram and choose **Unmark Activity > Default Activity**.

You should not specify a train stop in the middle of a train as a default activity (for more information, see [Section 17.6, "Creating a Train"](#)).

[Example 13–5](#) contains sample metadata for a default activity called SurveyPrompt in a task flow definition:

Example 13–5 Default Activity Metadata in Task Flow Definition

```
<task-flow-definition>
  <default-activity>SurveyPrompt</default-activity>
  <view id="SurveyPrompt">
    <page>/SurveyPrompt.jspx</page>
  </view>
</task-flow-definition>
```

13.5 Running ADF Task Flows

The procedure for running and debugging task flows differs depending on whether the task flow is bounded or unbounded, whether it contains pages or page fragments, or whether it accepts input parameters.

13.5.1 How to Run a Task Flow Definition That Contains Pages

You can run or debug an ADF bounded task flow that contains view activities that are pages.

For information on how to run a task flow definition that contains view activities that are page fragments, see [Section 13.5.2, "How to Run a Task Flow Definition That Uses Page Fragments"](#).

Note: You can select a view activity inside an ADF task flow diagram or the Application Navigator and choose **Run** to run an ADF bounded task flow.

In an ADF unbounded task flow, you must designate the view as a default activity and run the ADF unbounded task flow from the Application Navigator. For more information, see [Section 13.4, "Designating a Default Activity in an ADF Bounded Task Flow"](#).

If the first activity that runs in the ADF task flow is an activity type other than view, you must use an ADF bounded task flow.

To run or debug a task flow definition that uses pages:

- In the task flow diagram, right-click the task flow and choose either **Run** or **Debug**.
- You can also run the task flow directly by entering its URL in the browser, for example.
`http://mymachine:8988/StoreFrontModule-StoreFrontUI-context-root/faces/home`
- You can right-click the task flow definition in the Application Navigator and choose either **Run** or **Debug**.

13.5.2 How to Run a Task Flow Definition That Uses Page Fragments

ADF bounded task flows that use page fragments are intended to run only within an ADF region. A page fragment is a JSF JSP document that is rendered as content in another JSF page. For more information, see [Section 16.1.9, "What You May Need to Know About Page Fragments"](#).

To run or debug a task flow definition that uses page fragments:

1. Create a JSF page containing a region that is bound to the task flow definition. When you drop a bounded task flow containing page fragments onto a JSF page, JDeveloper does this automatically for you.
2. Create a view activity in the project's unbounded task flow that refers to the page. See [Section 13.2.2, "How to Add an Activity to an ADF Task Flow"](#) for more information.
3. Right-click the view activity in the Application Navigator or in the task flow diagram and choose **Run**.

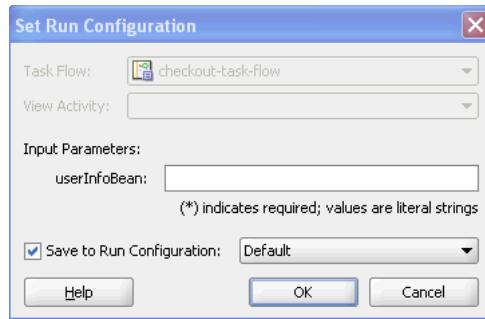
13.5.3 How to Run a Task Flow Definition That Has Parameters

Before you run a task flow definition with parameters, you must first run a task flow definition containing pages. For more information about bounded task flow input parameters, see [Chapter 15, "Using Parameters in Task Flows"](#).

To run a task flow definition that has input parameter definitions:

1. If the task flow definition has defined input parameters, the Set Run Configuration dialog displays after you select either **Run** or **Debug**, as shown in [Figure 13–17](#).

Figure 13–17 Set Run Configuration dialog



2. In the **Input Parameters** list, enter values that you want to be passed as input parameters to the task flow. If you do not specify a value, the input parameter is not used when calling the bounded task flow.

Each required input parameter in the list displays with an asterisk, as shown in [Figure 13–17](#). You must specify the parameter value as a literal string. You cannot specify an EL expression.

3. Click **OK**.

13.5.4 How to Run a JSF Page

You can run a JSF page by right-clicking the page in the Application Navigator and choosing **Run**. However, if the page contains navigation UI components such as a button or link, navigation is not guaranteed to work.

To run a JSF Page with fully functioning navigation:

1. Create a bounded or unbounded task flow. See [Section 13.2.1, "How to Create an ADF Task Flow"](#) for more information.
2. Add a view activity to the task flow. See [Section 13.2.2, "How to Add an Activity to an ADF Task Flow"](#) for more information.
3. In the Application Navigator, select the JSF page you want to run and drop it on top of the view activity in the task flow diagram.
This associates the view activity with the JSF page.
4. In the diagram, right-click the view activity and choose **Run**.

13.5.5 How to Run an ADF Unbounded Task Flow

To run or debug an unbounded task flow, you must select a specific view activity with which to start.

To run a view activity in an ADF unbounded task flow:

- In the task flow diagram, right-click the view activity and choose either **Run** or **Debug**.
The unbounded task flow runs beginning with the selected view activity.
- If you have selected something other than a single view activity (or have nothing selected), you are prompted to select one in the Set Run Configuration dialog.

13.5.6 How to Set a Run Configuration for a Project

A run configuration contains settings that determine how projects run, such as specifying the first activity to run in a task flow. You can define one or more run configurations for a project. Within a run configuration, you can designate an ADFc source file as the default run target. When you run the project, the source file is the first to run.

To define a default task flow run target:

1. Select the project in the Application Navigator.
2. From the main menu, choose **Run > Choose Active Run Configuration > Manage Run Configurations**.
3. Choose **Run/Debug/Profile** and choose **New**.
4. Enter a name for the new run configuration.
5. If you want to base the new configuration on an existing one, choose a configuration in the **Copy Configuration Settings** dropdown list.
6. Click **OK** to exit the dialog.
7. Click **Edit**.
8. In the **Default Run Target** dropdown list, select a source file for the ADF task flow that should run first when you run the project.

Once you choose a task flow, you can set a view activity (for an unbounded task flow) or input parameters (for task flow definitions).

9. In the left panel of the Edit Configuration dialog, click **ADF Task Flow**.
10. In the **Task Flow** drop-down list, located on the right panel, select the ADF task flow containing the run target.
11. If you are running an unbounded task flow, the Edit Run Configuration dialog displays the **Run Target Activity** list. Select the view activity that will run first in the application.
12. Click **Open**.

The next time you run the project, the saved run configuration will be available in the **Run > Choose Active Run Configuration** menu.

If you are running a bounded task flow that has been set up to accept input parameters, a dialog will display a section for specifying values for all input parameters defined for the bounded task flow (For more information, see [Chapter 13.5.3, "How to Run a Task Flow Definition That Has Parameters"](#)).

13.6 Setting Project Properties for ADF Task Flows

You can set a project property to change the default source directory for ADF task flows created within a project. The task flow definition for any ADF task flow you create within the project will be located in this directory. This source directory for a new project is, by default,

C:\JDeveloper\mywork\Application46\Project1\public_html\WEB-INF

To change the default source directory for task flow definitions:

1. In the Application Navigator, right-click on the project and choose **Project Properties**.
2. In the left panel of the Project Properties dialog, click **ADF Task Flows**.
3. Select **Use Project Settings**.
4. In the **default ADF Task Flow Source Directory** field enter a new default directory for task flow definitions created within the project.

13.7 Refactoring to Create New ADF Task Flows and Templates

You can convert existing activities, JSF page flows, and JSF pages into new ADF Controller components such as ADF bounded task flows and task flow templates.

13.7.1 How to Create an ADF Bounded Task Flow from Selected Activities

You can create a new ADF bounded task flow based on activities you select in an existing ADF bounded or unbounded task flow.

To create a new ADF bounded task flow from selected activities:

1. Open the ADF unbounded or bounded task flow containing the activities you want to use in the new task flow.
2. In the task flow diagram, select one or more activities.

Tip: To select multiple activities in a diagram, click the left mouse button and drag the cursor over the activities.

You can also press the Ctrl key while selecting each activity.

3. Right-click your selection and choose **Extract Task Flow**.

The Create Task Flow dialog displays, which allows you to create a new ADF bounded task flow. For more information, see [Section 13.2, "Creating Task Flows"](#).

When you are done, the new ADF bounded task flow displays in the editor. The properties shown in [Table 13-4](#) are automatically set for the new task flow.

Table 13-4 Properties Updated in the New ADF Bounded Task Flow

Property	Value
Task flow definition ID	Value you entered in the Task Flow ID field in the Create ADF Task Flow dialog.
Default activity	Determined as the destination of all incoming control flow cases. If more than one destination exists, an error is flagged and the entire operation is rolled back.

Table 13–4 (Cont.) Properties Updated in the New ADF Bounded Task Flow

Property	Value
Control flow rules	<p>Control flow cases with selected source activities are included in the new ADF bounded task. A source activity is an activity from which a control flow leads. The new ADF bounded task flow includes the following types of control flow cases:</p> <ul style="list-style-type: none"> ▪ Both the source and target activities in the control flow case were selected to create the new task flow. ▪ Only the source activity was selected to create the new task flow. Destinations are changed to the corresponding new task flow return activities added for each outcome.

The following changes automatically occur in the originating task flow (the task flow containing the activities you selected as the basis for the new task flow):

- A new task flow call activity is added to the originating task flow. The task flow call activity calls the new ADF bounded task flow.
- The selected activities are removed from the originating task flow.
- Existing control flow cases associated with the activities you selected are removed from the originating task flow. They are replaced with new control flow cases:
 - An incoming control flow case to the old activity is redirected to the new task flow call activity.
 - An outgoing control flow cases from the old activity is redirected from the new task flow call activity.

13.7.2 How to Create a Task Flow from JSF Pages

You can create a new ADF bounded task flow based on selected pages in a JSF page flow. Only pages that are part of a flow (that is, those that are linked by JSF navigation cases) are converted to view activities in the new task flow.

To create a new task flow from selected JSF pages in a page flow.

1. In the task flow editor, open the page flow containing the pages you want to use in the new bounded task flow.
2. In the task flow diagram, select one or more JSF pages.

Tip: To select multiple elements in a diagram, click the left mouse button and drag the cursor over the elements.

You can also press the Ctrl key while selecting each element.

3. Right-click your selection and choose **Generate ADF Task Flow**.

The Create Task Flow dialog displays, which allows you to create a new ADF unbounded or bounded task flow. For more information, see [Section 13.2, "Creating Task Flows"](#).

13.7.3 How to Convert ADF Bounded Task Flows

You can convert an existing ADF bounded task flow to an unbounded task flow or change whether the views it contains are pages or page fragments. [Table 13–5](#) describes the results of each conversion.

Table 13–5 Converting ADF Bounded Task Flows

Conversion	Result
ADF bounded task flow to unbounded task flow	ADF bounded task flow loses all metadata not valid for unbounded task flows, such as parameter definitions and transactions.
ADF bounded task flow to use JSF pages	Converts page fragments associated with any view activities in the task flow to JSF pages. Old page fragments are saved if you select the Keep Page Fragment checkbox. New JSF page names default to the name of the old page fragment.
ADF bounded task flow to use page fragments	Converts all pages associated with view activities in the ADF bounded task flow to page fragments. Old pages are saved if you select the Keep Page checkbox. New page fragment names default to the name of the old page

To convert an ADF bounded task flow:

1. In the task flow editor, open the bounded task flow diagram in the editor.
2. Right-click anywhere in the diagram other than on an activity or control flow.
3. Choose a menu item such as **Convert to Unbounded Task Flow** or **Convert to Task Flow with Page Fragments**.

If the bounded task flow contains fragments, the menu item will be **Convert to Task Flow with Pages**.

13.8 What You Should Know About Task Flow Constraints

[Table 13–6](#) summarizes assumptions about and constants for using task flows, activities, and other associated ADF Controller features.

Table 13–6 ADF Controller Features Assumptions and Constraints

Feature Area	Assumption/ Constraint	Description
ADF Controller objects and diagram UI	JSF view layer	ADF Controller operates only in a JSF 1.2 environment. Oracle's web-based Fusion web application strategy focuses on JSF as the sole view layer technology.
Dependent on Oracle ADF Faces		ADF Controller extensions are implemented on top of Oracle's ADF Faces. They are dependent on the ADF Faces libraries, but ADF Controller can run against any JSF implementation, providing these libraries are present.
Navigation and state management encapsulated		ADF Controller encapsulates both navigation and, to some extent, state management. JSF and the Servlet API are still available for the basic management of state at the application, session, and request levels.
Model layer		ADF model layer is used to implement the application's model layer.

Table 13–6 (Cont.) ADF Controller Features Assumptions and Constraints

Feature Area	Assumption/ Constraint	Description
	No supported migration path from struts or model 1	<p>There is no support for a migration from Struts or Model 1 to the Fusion ADF Controller.</p> <p>However, you can create a new ADF bounded task flow based on selected pages in a JSF page flow. For more information, see Section 13.7.2, "How to Create a Task Flow from JSF Pages".</p>
Bounded task flow	Exposed as pageFlow-scoped state	<p>ADF Controller manages implementation of a pageFlow-scoped state. Any auto-management functions provided by the framework, such as back button support and state cleanup function, assume pageFlow-scoped data. In order for an application to fully implement such functions for all of its pages, the entire application should be exposed as an ADF bounded task flow, using nested bounded task flows as needed. The application should store any state requiring versioning within the page flow scope.</p>
Page flow scope	Transactional boundaries	The developer will use ADF bounded task flows to manage transaction boundaries.
	Access availability within ADF lifecycle	<p>An application cannot attempt to access the pageFlow scope early in the ADF Lifecycle before ADF Controller is ready to provide it.</p>
		<p>Page flow scope is not guaranteed to be available for access until after Before and After listeners have executed on the Restore View phase. ADF Controller uses Before and After listeners on the Restore View phase to synchronize the server side state with the request. This is where things such as browser back-button detection and bookmark dereference are handled.</p>

14

Working with Task Flow Activities

This chapter describes how to use activities in your ADF task flows. The chapter contains detailed information about each task flow activity that displays in the Component Palette and its properties.

This chapter includes the following sections:

- [Section 14.1, "Introduction to Activity Types"](#)
- [Section 14.2, "Using View Activities"](#)
- [Section 14.3, "Using URL View Activities"](#)
- [Section 14.4, "Using Router Activities"](#)
- [Section 14.5, "Using Method Call Activities"](#)
- [Section 14.6, "Using Task Flow Call Activities"](#)
- [Section 14.7, "Using Task Flow Return Activities"](#)
- [Section 14.8, "Using Save Point Restore Activities"](#)
- [Section 14.9, "Using Parent Action Activities"](#)

14.1 Introduction to Activity Types

An *activity* represents a piece of work that is performed when the task flow runs. It displays in the ADF task flow editor as a node. You can add most activities to both ADF bounded and unbounded task flows, although some activity types can be added only to an ADF bounded task flow.

The bounded task flow shown in [Figure 14–1](#) contains activities that run in order to check out of the application:

1. A call to a method synchronizes the items a user may have chosen before logging in with those selected after logging in
2. A page (view activity) that displays the items the user has currently selected and another page that summarizes the order
3. An activity that causes control to return back to the calling unbounded task flow shown in [Figure 14–1](#)

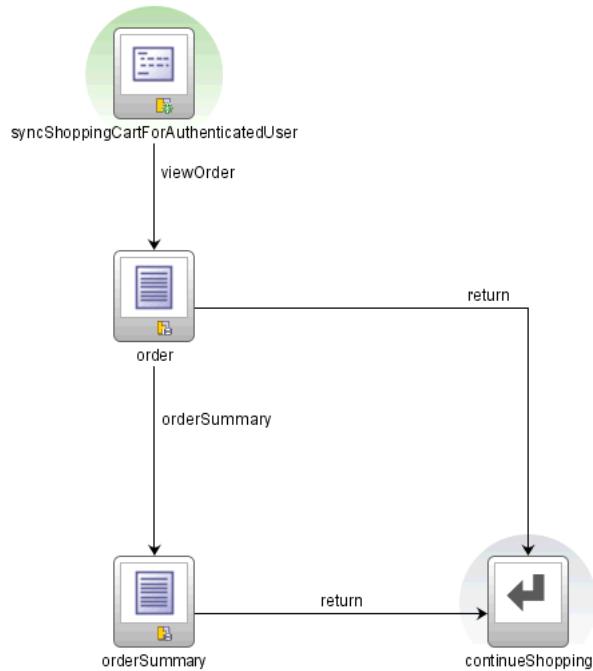
Figure 14–1 Checkout Bounded Task Flow in Fusion Order Demonstration Application

Table 14–1 describes the types of activities and control flows you can add to a task flow.

Table 14–1 ADF Task Flow Activities and Control Flows

Icon	Component Name	Description
	Method Call	Invokes a method, typically a method on a managed bean. A method call activity can be placed anywhere within an application's control flow to invoke application logic based on control flow rules. See Section 14.5, "Using Method Call Activities" for more information.
	Router	Evaluates an EL expression and returns an outcome based on the value of the expression. For example, a router in a credit check task flow might evaluate the return value from a previous method call and generate success, failure, or retry outcomes based on various cases. These outcomes can then be used to route control to other activities in the task flow. See Section 14.4, "Using Router Activities" for more information.
	Save Point Restore	Restores a previous persistent save point, including application state and data, in an application supporting save for later functionality. See Section 17.5, "Saving for Later" for more information.
	Task Flow Call	Calls an ADF bounded task flow from an ADF unbounded task flow or another bounded task flow. See Section 14.6, "Using Task Flow Call Activities" for more information.

Table 14–1 (Cont.) ADF Task Flow Activities and Control Flows

Icon	Component Name	Description
	Task Flow Return	Identifies when a bounded task flow completes and sends control flow back to the caller. (Available for ADF bounded task flows only). See Section 14.7, "Using Task Flow Return Activities" for more information.
	URL View	Redirects the root view port (for example, a browser page) to any URL-addressable resource, even from within the context of an ADF region. See Section 14.3, "Using URL View Activities" for more information.
	View	Displays a JSF page or page fragment. Multiple view activities can represent the same page or same page fragment. See Section 14.2, "Using View Activities" for more information. See Section 18.3, "Creating a Web Page" for more information about pages and page fragments.
	Control Flow Case	Identifies how control passes from one activity to the next in the application. See Section 13.1.3, "Control Flows" for more information.
	Wildcard Control Flow Rule	Represents a control flow case that can originate from any activities whose ids match a wildcard expression. For example, it can represent a control case <code>from-activity-id</code> containing a trailing wildcard such as <code>foo*</code> . See Section 13.2.4, "How to Add a Wildcard Control Flow Rule" for more information.
	Parent Action	Allows an ADF bounded task flow to generate outcomes that are passed to its parent view activity. See Section 14.9, "Using Parent Action Activities" for more information.

[Table 14–2](#) describes the annotations (notes and attachments) you can add to an ADF task flow.

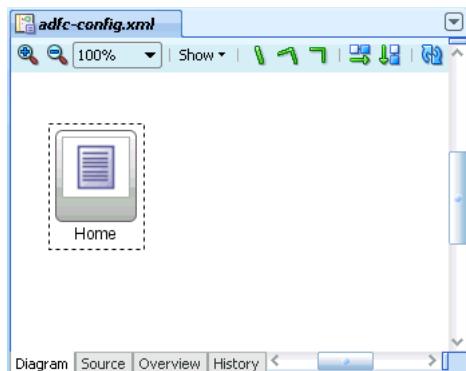
Table 14–2 ADF Task Flow Diagram Annotations

Icon	Icon Name	Description
	Note	Adds a note to the task flow diagram. You can select the note in the diagram to add or edit text.
	Note Attachment	Attaches an existing note to an activity or a control flow case in the diagram.

14.2 Using View Activities

The primary type of task flow activity is a view, which displays a JSF page or page fragment. A page fragment is a JSF JSP document that is rendered as content in another JSF page. Page fragments are typically used in bounded task flows. The bounded task flow can be added to a page as region. For more information, see [Section 16.1, "Introduction to ADF Regions"](#).

[Figure 14–2](#) shows the Home view activity, located in the Fusion Order Demo application.

Figure 14–2 View Activity

A view activity is associated in metadata with a physical JSF page or page fragment. The view activity is identified by an `id` attribute. The page or page fragment name is identified by a `<page>` element in the task flow metadata:

```
<view id="Home">
  <page>/Home.jspx</page>
</view>
```

The view activity ID and page name do not have to be the same.

The file extension for a page fragment is `.jsff`:

```
<view id="Home">
  <page>WEB-INF/Home.jsff</page>
</view>
```

14.2.1 Adding a View Activity

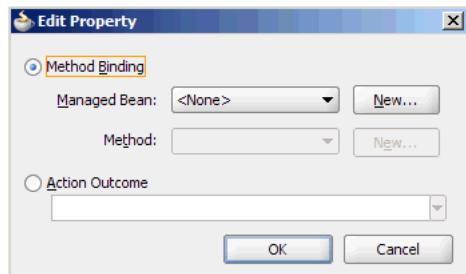
The steps for adding a view activity are similar to those for adding any activity to a task flow diagram (see [Section 13.2.2, "How to Add an Activity to an ADF Task Flow"](#)). After you add the view activity, you can double-click it to display the Create JSF JSP Page wizard, which enables you to create a new page or page fragment. You also use the wizard to define characteristics for the page or page fragment. JDeveloper automatically associates the completed page or page fragment with the view activity.

You can also drag an existing page or page fragment from the Application Navigator and drop it on top of a view activity.

If you drag a page or page fragment to any other location on the diagram, a new view activity associated with the page or page fragment is automatically created. During creation, a default `id` for the view activity is automatically generated (for example `Home`) based on the name of the page or page fragment.

14.2.2 Transitioning Between View Activities

Transitioning refers to one view activity passing control to another view activity. For example, control flow can be initiated at runtime by selecting a UI component on a page, such as a button or link. The **Action** attribute of the UI component should be set to the corresponding control flow case `from-outcome` leading to the next task flow activity. You can navigate from a view activity to another activity using either a constant or dynamic value on the **Action** attribute of the UI component.

Figure 14–3 Edit Property dialog

- Constant: value of the **Action** attribute of the component is an Action Outcome, as shown in [Figure 14–3](#). Action Outcome is a constant value that always triggers the same control flow case. When an end user clicks the component, the activity specified in the control flow case is performed. There are no alternative control flows.
- Dynamic: value of the **Action** attribute of the component is bound to a managed bean or a method. The value of the method binding determines the next control flow case that should be performed.

For example, the method might verify user input on a page and return one value if the input is valid and another value if the input is invalid. Each of these different action values could trigger different navigation cases, causing the application to navigate to one of two possible target pages.

For more information about components that are bound to data control methods, see [Section 26.2, "Creating Command Components to Execute Methods"](#).

14.2.2.1 How to Transition to a View Activity

Before you begin, you should already have a target view activity, as well as a JSF page on which you will add a component. The component's action will be based on the **from-outcome** of the control flow case leading to the target activity.

To transition to a view activity:

1. Add a UI component to the JSF page using one of the following techniques:
 - Open the JSF page. From the **ADF Faces Common Components** list in the Component Palette, drag a navigation UI component such as a button or link onto the JSF page.
 - From the Data Controls panel, drag and drop an operation or a method onto the JSF page and choose **Rich Command Button** or **Rich Command Link** from the context menu.
2. Select the UI component and open the Property Inspector.
3. On the Common page, expand the **Button Action** section.
4. From the dropdown menu next to **Action**, and choose **Edit**.
5. Select **Action Outcome**.
6. From the **Action Outcome** dropdown list select a value.

The list contains control flow case **from-outcomes** already defined for the view activity associated with the page.

Tips: The action attribute of the UI component can be bound either to a literal string to hardcode a navigation case, or it can be bound to a method binding expression that points to a method, which takes no arguments and returns a String. It can't be bound to any other type of EL expression.

7. Click **OK**.

14.2.2.2 What Happens When You Transition Between Activities

[Example 14–1](#) contains an example of a control flow case defined in the XML source file for an ADF bounded or unbounded task flow.

Example 14–1 Control Flow Case Defined in XML Source File

```
<control-flow-rule>
    <from-activity-id>Start</from-activity-id>
        <control-flow-case>
            <from-outcome>toOffices</from-outcome>
            <to-activity-id>WesternOffices</to-activity-id>
        </control-flow-case>
    </control-flow-rule>
```

As shown in [Example 14–2](#), a button on a JSF page associated with the Start view activity specifies `toOffices` as the **action** attribute. When the user clicks the button, control flow passes to the `WesternOffices` activity specified as the `to-activity-id` in the control flow metadata.

Example 14–2 Static Navigation Button Defined in a View Activity

```
<af:commandButton text="Go" action="toOffices">
```

14.2.3 Bookmarking View Activities

Bookmarking is available only for view activities within ADF unbounded task flows.

When an end user bookmarks a page associated with a view activity, the URL that displays in the browser's address field for the view is saved as the bookmark. In most cases, this URL cannot be used to redisplay the page associated with the view. For example the URL may contain Windows state information that cannot be used to redisplay the page.

The bookmark URL should contain information that enables dynamic content on the page to be reproduced. For example, if an end user bookmarks a page displaying a customer's contact information, the bookmark URL needs to contain not only the page but also some identifier for the customer. This will enable contact information for the same customer to display when he returns to the page using the bookmark.

To ensure that the URL for a page displayed in a browser can be used as a bookmark, identify the view activity associated with the page as bookmarkable.

At runtime, you can identify if a view activity within an unbounded task flow has been designated as bookmarkable using the `ViewBookmarkable()` method. The method is located off the `ViewPortContext`.

After you designate a view activity as bookmarkable, you can optionally specify one or more URL Parameters. The value of `url-parameter` is an EL expression. The EL expression specifies where the parameters that will be included in the URL are retrieved when the bookmarkable URL is generated. The EL expression also stores a

value from the URL when the bookmarkable URL is dereferenced. The converter option identifies a method that performs conversion and validation when parameters are passed via bookmarkable view activity URLs.

In addition, you can 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.

Instead of designating the view activity as bookmarkable, you can specify the redirect option. redirect causes the ADF Controller to create a new browser URL for the view activity. The original URL for the view activity is no longer used. For more information, see [Section 14.2.3.2, "How to Specify HTTP Redirect"](#) for more information.

[Example 14–3](#) contains the URL syntax for a bookmarked view activity.

Example 14–3 ADF Unbounded Task Flow View Activity URL Syntax

```
<server root>/<app_context>/faces/<view activity id>?<param name>=<param value>&...
```

The syntax of the URL for the bookmarked view activity is:

- <server root>: Provided by customization at site or admin level, for example, `http://mycompany.com/internalApp`.
- <app context>: The web application context root, for example, `myapp`. The context root is the base path of a web application. For example, <app_context> maps to the physical location of the WEB-INF node on the server.
- faces: The faces servlet mapping. The value in faces points to the node containing the `faces-config.xml` configuration file.
- <view activity id>: The identifier for the bookmarked view activity, for example, `edit-customers`.
- <param name>: The name of the bookmarked view activity URL parameter, for example, `customer-id`.
- <param value>: The parameter value, derived from an EL expression, for example, `# {pageFlowScope.employee.id}`. The value of the EL expression must be capable of being represented as a string.

[Example 14–4](#) contains a sample URL for a bookmarkable view activity in an ADF unbounded task flow.

Example 14–4 Sample URL for Bookmarkable View Activity

```
http://mycompany.com/internalApp/MyApp/faces/edit-customers?customer-id=1234&...
```

14.2.3.1 How to Create a Bookmarkable View Activity

To create a bookmarkable view activity, designate a view activity as bookmarkable, specify a URL parameter in the bookmark, and specify a method that is executed after the bookmark is dereferenced.

To designate a view activity as bookmarkable:

1. In the unbounded task flow diagram, select the view activity.
2. In the Property Inspector, click **Bookmark**.

3. In the **bookmark** dropdown list, select **true**.
4. Expand the **URL Parameters** section to add optional URL parameters that will be included in the URL for the bookmarked view activity:
 - **name**: A name for the parameter.
 - **value**: A settable EL expression that, when evaluated, specifies the parameter value, for example, `# {pageFlowScope.employeeID}`. The value must be capable of being represented as a string.
 - **converter**: (optional): An EL expression to an object that implements `oracle.adf.controller.URLParameterConverter`.

The **value** is where the parameters that will be included in the URL are retrieved from when the bookmarkable URL is generated. In addition, parameters are stored here when the bookmarkable URL is dereferenced.

If the EL expression entered in **value** returns `NULL`, the parameter is omitted from the bookmarked view activity URL.

The **name** and **value** are used to append a bookmark parameter to the view activity URL, as shown in [Example 14-4](#).

5. In the **converter** field, you can enter an optional value binding to use for each bookmark URL parameter value, for example,
`# {pageFlowScope.employee.idConverter}`.

A URL parameter converter's `getAsObject()` method takes a single string value as its input parameter and returns an object of the appropriate type. ADF Controller invokes the converter method on the URL parameters before applying the parameter value to the application's model objects. Similarly, the converter's `getAsString()` method takes an object as its input parameter and returns a string representation that is used on the URL.

In a JSF application, data values are converted and validated using the converters and validators specified with the UI components on the submitting page. In a Fusion web application using a bookmark URL, there is no submitting page to handle the conversion and validation. Therefore, you have the option of designating a converter to use for each URL parameter.

14.2.3.2 How to Specify HTTP Redirect

The `redirect` option specified for a view activity indicates that ADF Controller should issue an HTTP redirect for a view activity request. The redirected request creates a new browser URL for the view activity. The original view URL is no longer used.

When specified, the redirect will occur from a client GET request. For HTTP GETs, the `# {bindings}` EL scope is invalid until the ADF Controller and the ADF Model layer set up a new bindings context for the page. Therefore, the redirected input parameter for the view activity cannot be mapped.

A view activity can be identified as either bookmarkable or identified with the `redirect` option, but not both.

Note: If you want `http://www.mycompany.org/x.html` to instead display what is at `http://www.mycompany.org/y.html`, do not use refresh techniques such as:

```
<META HTTP-EQUIV=REFRESH CONTENT="1;
URL=http://www.example.org/bar">
```

This technique could adversely affect back button behavior. If an end user clicks a browser Back button, the refresh occurs again, and navigation is forward, not backward as expected.

In this situation, use HTTP redirect instead.

To specify HTTP redirect for a view activity:

1. In the unbounded task flow diagram, select the view activity.
2. In the Property Inspector, click **Common**.
3. In the **redirect** dropdown list, select **true**.

14.2.3.3 What Happens When You Designate a View as Bookmarkable

When you designate a view activity as bookmarkable, a bookmark element is added to the metadata for the view activity, as shown in [Example 14–5](#). The bookmark element can optionally contain metadata specifying URL parameters and a method that is executed after the bookmark is dereferenced.

Example 14–5 Sample Metadata for a Bookmarkable View Activity

```
<view id="employee-view">
  <page>/folderA/folderB/display-employee-info.jspx</page>
  <bookmark>
    <url-parameter>
      <name>employee-id</name>
      <value>#{pageFlowScope.employee.id}</value>
      <converter>#{pageFlowScope.employee.validateId}</converter>
    </url-parameter>
    <method>#{pageFlowScope.employee.queryData}</method>
  </bookmark>
</view>
```

14.3 Using URL View Activities

You can use a URL view activity to redirect the root view port (for example, a browser page) to any URL-addressable resource, even from within the context of an ADF region. URL addressable resources include:

- ADF bounded task flows
- View activities in an ADF unbounded task flow
- Addresses external to the current web application (for example, Google)

To display the resource, you must specify an EL expression that is evaluated at runtime to generate the URL to the resource. In addition, you can specify EL expressions that, when evaluated, are added as parameters and parameter values to the URL.

A URL view activity redirects the client regardless of the view port (root view port or an ADF region) from which it is executed. The `<redirect>` element of a view activity performs in a similar way, except that it can be used only if the view activity is within the root view port. The `<redirect>` element is ignored within the context of an ADF region. For more information, see [Section 14.2.3.2, "How to Specify HTTP Redirect"](#).

Redirecting elsewhere within the same application using URL view activities (not the `<redirect>` element) is handled similarly to back button navigation since the task flow stack is cleaned up. Redirecting out of the web application is handled like dereferencing a URL to a site external to the application.

To add a URL view activity to a task flow diagram.

1. Drag a URL view activity from the ADF Task Flow section in the Component Palette onto the diagram.
2. In the task flow diagram, select the URL view activity.
3. On the Common page of the Property Inspector, enter an id, for example, `externalSite`.
4. Click the button next to the `url` field and create an EL expression, for example, `# {pageFlowScope.someBean.redirectURL}`.
The EL expression is evaluated at runtime to generate the URL to the resource.
5. Expand the **URL Parameters** section to add optional URL parameters that will be included in the URL:
 - **name:** A name for the parameter.
 - **value:** An EL expression that, when evaluated, generates the parameter value.
 - **converter:** A settable EL expression that, when evaluated, specifies a method to perform conversion and validation when parameters are passed via bookmarkable view activity URLs. For more information, see [Section 28.2.4, "How to Enable ADF Authentication and Authorization"](#).

14.3.1 Constructing a URL for Use Within a Portlet

When constructing a URL for use in a task flow's URL view activity that may be used within the context of a portlet, construct the URL by calling one of the following:

- `ControllerContext.getLocalViewActivityURL()`
- `ControllerContext.getGlobalViewActivityURL()`, passing in the target `viewId`

or a fully qualified absolute URL, a context path relative URL, or a URL that is relative to the current view.

Note: If you call the `ControllerContext.getLocalViewActivityURL()` or `ControllerContext.getGlobalViewActivityURL()` methods to construct the redirect URL, do not call `ExternalContext.encodeActionURL()` with the response before calling `ExternalContext.redirect()`.

This is because the methods already incorporate the necessary encoding of the URL.

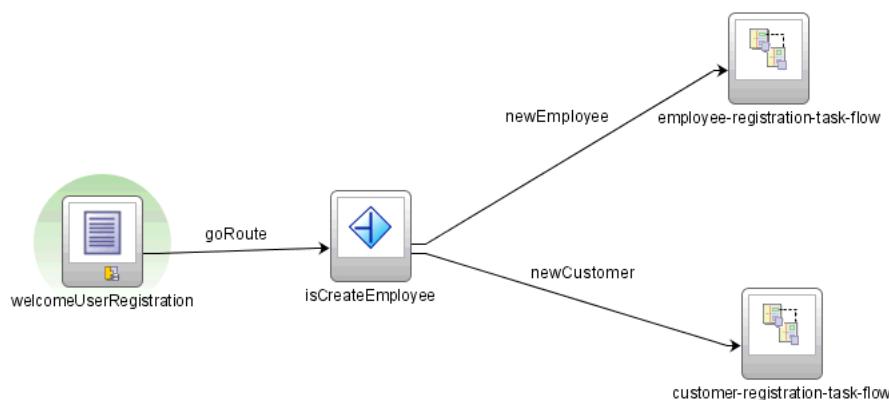
When a URL view activity is used within a task flow in a portlet, the following behavior occurs:

- If the redirect URL refers to a location within the portlet application and doesn't contain a `queryString` parameter named `x_DirectLink` whose value is `true`, then the portlet within the containing page will navigate to this new view.
- Otherwise, a client redirect is issued, resulting in the user being directed away from the application or containing page and to the URL.

14.4 Using Router Activities

You can use a router activity to declaratively route control to activities based on logic specified in an EL expression. As shown in [Figure 14–4](#), a router has multiple control flows leading from it to different activities.

Figure 14–4 Router for Alternate Control Flow Cases



Each control flow can correspond to a different router case. Each router case contains the following elements, which are used to choose the activity to which control is next routed:

- **expression:** An EL expression evaluating to either `true` or `false`, for example, `# { (pageFlowScope.welcomeUserRegistrationBean.userSelection eq 'Customer') }`

The first expression that evaluates to `true` is used to determine the corresponding outcome.

- **outcome:** A value returned by the router activity if the EL expression evaluates to `true`, for example, `newCustomer`.

If the router **outcome** matches a **from-outcome** on a control flow case, control passes to the activity that the control flow case points to. If none of the cases for the router activity evaluates `true`, or if no cases are specified, the outcome specified in the router **default outcome** field (if any) is used.

For example, suppose you want to base control flow on whether a user clicks the **Create a New Customer** or **Create a New Employee** button on the `welcomeUserRegistration` page fragment shown in [Figure 14–4](#).

You could add an EL expression for one of the router cases that evaluates whether the user entered in the input text field on the user registration page fragment is a new customer. You would next specify an expected **outcome**, for example, `newCustomer`. As shown in [Figure 14–4](#), if the expression evaluates to `true`, control passes to the `customer-registration-task-flow` task flow call activity, based on the control flow case **from-outcome**, `newCustomer`.

Best Practice:

If your routing condition can be expressed in EL, use a router.

Using a router allows you to do more when you are designing the ADF task flow that contains it. The router activity allows you to show more information about the condition on the ADF task flow, thus making it more readable and useful to someone else who looks at your diagram.

Using a router activity also makes it easier to modify your application later. For example, you may want to modify a routing condition later or add an additional routing condition

To define a control flow using the router activity:

1. From the ADF Task Flow page of the Component Palette, drag a Router activity to the task flow diagram.
2. In the task flow diagram, select the router activity.
3. From the main menu, choose **View -> Property Inspector**.
4. On the Common page of the Property Inspector, enter an **id**.

The id is an identifier that is used to reference the router activity within the metadata, for example, `router1`.

5. Click the Add icon next to **Cases**.
6. Specify values 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 evaluating to `true` or `false`.

The expression can reference an input text field in a view activity. For example, suppose the value of the field is `# {pageFlowScope.value}`. The expression could be `# {pageFlowScope.value=='view2'}`, meaning that the specified **outcome** will be returned if a user enters `view2` in the field.

- **outcome:** Returned by the router activity when its corresponding expression evaluates true.

You must account for each outcome with a matching control flow case or a wildcard control flow rule in your task flow diagram. For example, for each case **outcome**, you can ensure there is a corresponding **from-outcome** specified for a control flow case element leading from the router activity in the diagram. In [Figure 14–4](#), the value for both the case **outcome** and the control flow case element **from-outcome** is `newCustomer`. This ensures that control flow will pass to the `newCustomer` activity, the target of the control flow element.

7. In the Property Inspector, enter a **default-outcome**.

This outcome is returned if none of the cases for the router activity evaluates true, or if no cases are specified.

Example 14–6 identifies a default outcome, toRegion3. Control flow goes to the case whose from outcome is toRegion3 is returned by the router activity when none of its cases evaluates to true.

Example 14–6 Router Metadata Defining a Default Outcome

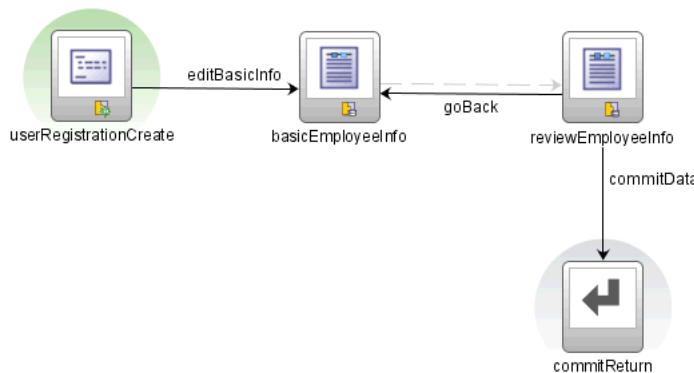
```
<router id="Router1">
    <case>
        <expression>#{binding.Region.InputValue='1'}</expression>
        <outcome>toRegion1</outcome>
    </case>
    <case>
        <expression>#{binding.Region.InputValue='2'}</expression>
        <outcome>toRegion2</outcome>
    </case>
    <case>
        <expression>#{binding.Region.InputValue='3'}</expression>
        <outcome>toRegion3</outcome>
    </case>
    <default-outcome>toRegion3</default-outcome>
</router>
```

14.5 Using Method Call Activities

In a standard JSF application, application logic can be invoked only from actions specified within the JSF page markup. A method call activity allows 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.

As shown in [Figure 14–5](#), the Fusion Order Demo application uses a method call activity in the Employee Registration bounded task flow. The activity calls userRegistrationCreate, a method exposed on the StoreServiceAM data control.

Figure 14–5 Method Call Activity in employee-registration-task-flow



You can set an outcome for the method that specifies a control flow case to pass control to after the method finishes. For more information, see [Section 13.1.3, "Control Flows"](#). You can specify the outcome as either:

- **fixed-outcome**: On successful completion, the method always returns this single outcome, for example, `success`. If the method doesn't complete successfully, an outcome isn't returned. If the method type is `void`, you must specify a `fixed-outcome`, not a `to-string`.
- **to-string**: If specified as true, the outcome is based on calling the `toString()` method on the Java object returned by the method. For example, if `toString()` returns `editBasicInfo`, navigation goes to the `editBasicInfo` control flow case shown in [Figure 14–5](#).

As shown in [Example 14–7](#), the method outcome and the method result are two different values. The `<return-value>` element specifies where to put the result of the `calculateSalesTax` method. The `<outcome>` element indicates which control flow case to use after the method finishes.

Example 14–7 Method Call Activity Metadata with Return and Outcome Elements

```
<method-call id="calculateSalesTax">
    <method>#{pageFlowScope.taxService.calculateSalesTax}</method>
    <return-value>#{pageFlowScope.result}</return-value>
    <outcome>
        <fixed-outcome>gotoError</fixed-outcome>
    </outcome>
</method-call>
```

Best Practice:

You can use a method call on an ADF task flow to invoke a method before a page renders, or you can use an `invokeAction` on a page definition.

If you want your method to execute before the page is rendered, it is usually best to use a method call activity in the task flow diagram rather than an `invokeAction` in the page definition file. By adding your method as a method activity on a page flow diagram, it is easier to invoke logic between pages. This allows you to do more at the time you're designing the task flow. You can also show more information on the task flow, thus making it more readable and useful to someone else who looks at your diagram.

You might want to use an `invokeAction` instead of a method call for one of the following reasons:

- You want the method to be executed in more than one phase of the page's lifecycle.
- You plan to reuse the page and page definition file, and want the method to be tied to the page.
- You are not using ADF Controller.

14.5.1 How to Add a Method Call Activity

Before you begin, you should have already created an ADF bounded or unbounded task flow. For more information, see [Section 13.2, "Creating Task Flows"](#). Drag a

method call activity from the Component Palette to the task flow diagram. You can associate the method call activity with an existing method by dropping a data control operation from the Data Controls panel directly onto the method call activity in the task flow diagram.

In the Fusion Order Demo, for example, you could drag the `addItemToCart` method from `StorefrontModelDataControl` to the diagram. Or you could drag a `setCurrentRowWithKey` or `setCurrentRowWithKeyValue` operation to the diagram from the Data Control Iterator to display or select the current row in a table.

Note: Parameters for data control method parameters are defined in the page definition for the corresponding page rather than within ADF Controller metadata. For more information, see [Section 26.3, "Setting Parameter Values Using a Command Component"](#).

You can also drag methods and operations directly on the task flow diagram. A new method call activity is created automatically after you drop it on the diagram. The following steps show how to specify an EL expression and other options for the method.

To add a method call activity to an ADF task flow:

1. In the Component Palette, drag a method call activity from the ADF Task Flow page to the diagram for the ADF task flow.

The method call activity optionally displays a default id, `methodCall1`, and a warning icon that indicates that a method EL expression has not yet been specified.



methodCall1

For more information about turning on the warning icons, see [Section 13.2.2, "How to Add an Activity to an ADF Task Flow"](#).

2. If you want to change the default ID, click the text that appears under the method call activity in the task flow diagram.

You can enter a name for the method call, for example, `addItemToCart`.

3. In the task flow diagram, select the method call activity.
4. On the Common page of the Property Inspector, enter an EL expression for the method in the **method** field.

For example, you can enter an EL binding expression such as
`#{bindings.addItemToCart.execute}`.

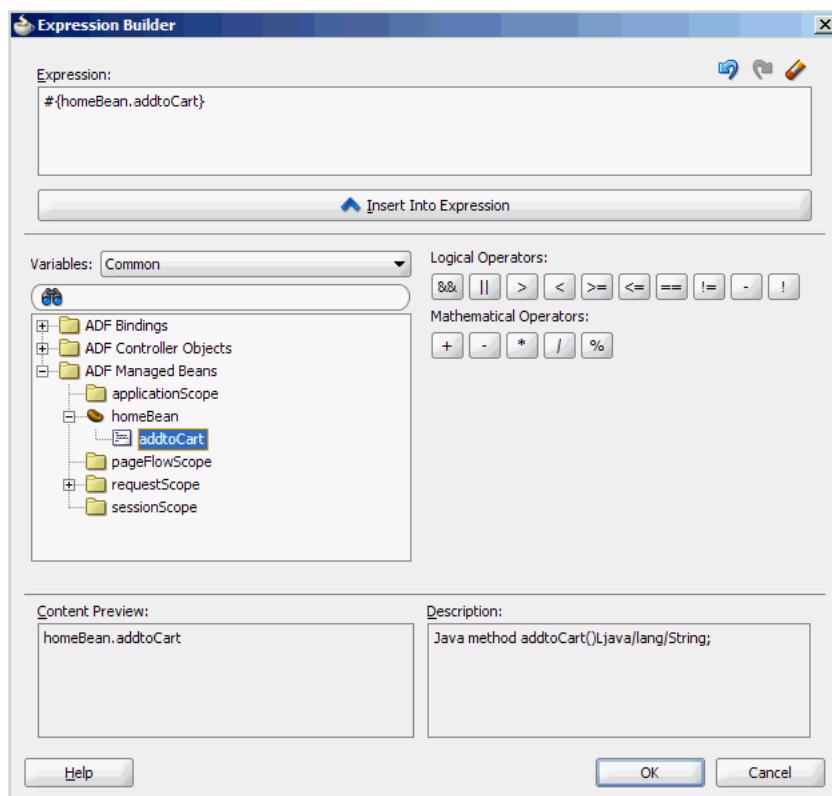
Note: The `bindings` variable in the EL expression indicates an ADF model binding from the current binding container. In order to specify the `bindings` variable, a binding container definition or page definition must be specified. See [Section 11.6, "Working with Page Definition Files"](#).

You can also use the Edit Property dialog box shown in [Figure 14–6](#) to build the EL expression for the method:

- a. In the Common page of the Property Inspector, from the drop-down menu next to the **method** field, choose **Expression Builder**.
- b. In the Expression Builder dialog, expand a node, for example, **ADF Bindings** and choose a method. Or, under the **ADF Managed Beans** node, navigate to the managed bean containing the method you want to call and select the method.
- c. Click **Insert Into Expression**.

When you are finished, the dialog should look similar to [Figure 14–6](#).

Figure 14–6 EL Expression for Method in Edit Property Dialog

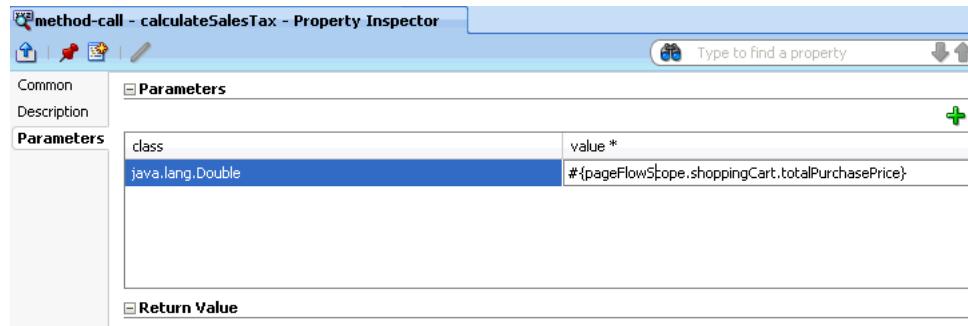


- d. Click **OK**.
5. In the Common page of the Property Inspector, expand the **Outcome** section and specify one of the following:
 - **fixed-outcome**: On successful completion, the method always returns this single outcome, for example, `success`. If the method doesn't complete successfully, an outcome isn't returned. If the method type is `void`, you must specify a **fixed-outcome**, not a **to-string**.
 - **to-string**: If you select `true`, the outcome is based on calling the `toString()` method on the Java object returned by the method.

14.5.2 How to Specify Method Parameters and Return Values

Figure 14–7 shows a single parameter defined for a method called calculateSalesTax. The **value** field contains an EL expression that evaluates to the parameter value.

Figure 14–7 Method Parameters in Property Inspector



If parameters haven't already been created by associating the method call activity to an existing method, follow the steps below.

To add method parameters:

1. Follow the steps in [Section 14.5.1](#) to add a method call activity to a task flow diagram.
2. In the task flow diagram, select the method call activity.
3. In the Property Inspector, click **Parameters**.
4. On the Parameter page, expand the **Parameters** section.
5. Click the plus (+) icon.
6. In the **class** field, enter the parameter class, for example, `java.lang.Double`.
7. In the **value** field, enter an EL expression indicating where the value for the parameter will be retrieved, for example,
`# {pageFlowScope.shoppingCart.totalPurchasePrice}`.

Tip: You can click the icon next to the **value** field and choose **Expression Builder** to search for the method parameters.

8. In the **return-value** field, enter an EL expression indicating where to store the method return value, for example, `# {pageFlowScope.Return}`.
9. Click **OK**.
10. Repeat the above steps to add additional parameters.

14.5.3 What Happens When You Add a Method Call Activity

[Example 14–8](#) shows how a method call to `userRegistrationCreate` appears in the XML source file for an ADF bounded task flow.

Example 14–8 Call to userRegistrationCreate method

```
<method-call id="userRegistrationCreate">
    <method>#{bindings.userRegistrationCreate.execute}</method>
    <outcome>
```

```
<fixed-outcome>editBasicInfo</fixed-outcome>
</outcome>
<method-call>
```

14.6 Using Task Flow Call Activities

You can use the task flow call activity to call an ADF bounded task flow from either an ADF unbounded or bounded task flow. Options on the task flow call activity allow you to call a bounded task flow located within the same application or a different application.

The called bounded task flow executes beginning with its default activity. There is no depth limit to the number of ADF bounded task flow calls. A called ADF bounded task flow can call another ADF bounded task flow, which can call another and so on.

To pass parameters into task flows, you must specify input parameter values on the task flow call activity. These values must correspond to the input parameter definitions on the called task flow definition (see [Section 14.6.2, "How to Specify Input Parameters on a Task Flow Call Activity"](#) for more information).

- The value on the task flow call activity **Input Parameter** specifies where the value will be taken from within the calling task flow.
- The value on the **Input Parameter Definition** for the called task flow specifies where the value will be stored within the called task flow definition once it is passed.

Tip: When a task flow definition is associated with a task flow call activity, input parameters are automatically inserted on the task flow call activity based on the input parameter definitions defined on the task flow definition. Therefore, the application developer only needs to assign values to the task flow call activity input parameters.

By default, all objects are passed by reference. Primitive types (for example, int, long, or boolean) are always passed by value.

The technique for passing return values out of the ADF bounded task flow to the caller is similar to the way that input parameters are passed. See [Section 15.4, "Specifying Return Values"](#) for more information.

14.6.1 How to Call an ADF Bounded Task Flow

To call an ADF bounded task flow, add a task flow call activity to the diagram for the calling bounded or unbounded task flow.

To call an ADF bounded task flow:

1. In the editor, open the task flow diagram for the calling ADF task flow.
2. In the ADF Task Flow dropdown list of the Component Palette, select a task flow call activity and drop it on the diagram.
3. Identify the called task flow using one of the following techniques:
 - In the task flow diagram, double-click the task flow call activity. The Create ADF Task Flow dialog displays, where you specify options for creating a new bounded task flow.

- Drag an existing ADF bounded task flow from either the Application Navigator or the Resource Palette and drop it on the task flow call activity in the task flow diagram.

Tips: You can also drag an ADF bounded task flow from the Application Navigator and drop it directly on the task flow diagram. This automatically adds to the diagram a task flow call activity that calls the ADF bounded task flow.

You can drop an ADF bounded task flow on a page or page fragment. If the ADF bounded task flow consists of pages (not page fragments), you can choose whether to add a Go Link or Go Button UI component on the page where you drop the task flow. An end user can click the button or link to call the task flow definition. This may in turn automatically generate the task flow call activity if the page is already associated with an existing view activity in an ADF task flow.

You cannot drop an ADF bounded task flow contained in one application to a diagram contained in another application using the Application Navigator, even though both applications appear in the navigator. In addition, you cannot drop a bounded task flow contained in one project onto a task flow diagram contained in another project.

Instead, you can package the bounded task flow in an ADF Library, then reuse it in your current application or project. You can then drag the bounded task flow from the Resource Catalog or from the Component Palette page that is created when you import the library. See [Section 31.1.2, "Using the Resource Palette"](#) for more information.

- Associate the task flow call activity to the called task flow definition by following these steps:
 - a. In the task flow diagram, select the task flow call activity.
 - b. On the Common page of the Property Inspector, expand the **Task Flow Reference** node.
 - c. Enter a **document** name.

This is the name of the source file containing the **id** of the called ADF bounded task flow, for example, `called-task-flow-definition.xml`.

- d. Enter an **id**.

This is the task flow definition id contained in the XML source file for the called ADF bounded task flow, for example, `targetTaskFlow`.

14.6.2 How to Specify Input Parameters on a Task Flow Call Activity

The suggested method for mapping parameters between a task flow call activity and its called bounded task flow is to first specify input parameter definitions for the called task flow definition. Then you can drag the task flow definition from the Application Navigator and drop it on the task flow call activity. The task flow call activity input parameters will be created automatically based on the task flow definition's input parameter definition. For more information, see [Section 15.2, "Passing Parameters to an ADF Bounded Task Flow"](#).

You can, of course, first specify input parameters on the task flow call activity. Even if you have defined them first, they will automatically be replaced based on the input parameter definitions of the called task flow definition, once it is associated with the task flow call activity.

If you haven't yet created the called task flow definition, you may still find it useful to specify input parameters on the task flow call activity. Doing so at this point allows you to identify any input parameters you expect the task flow call activity to eventually map when calling a task flow definition.

To specify input parameters on the task flow call activity

1. Select the task flow call activity in the task flow diagram and open the Property Inspector.
2. Click **Parameters**.
3. Click the + icon and enter a **name** for the parameter, for example, `empno`.

Tip: Dropping an ADF bounded task flow on a task flow call activity in a diagram automatically populates the **name** field.

4. Enter a parameter **value**, for example,
`# {pageFlowScope.callingTaskflowParm}`.
The **value** specifies where the parameter value will be taken from within the calling task flow.
By default, all objects are passed by reference. Primitive types (for example, int, long, or boolean) are always passed by value.
5. After you have specified an input parameter, you can specify a corresponding input parameter definition for the called ADF bounded task flow. For more information, see [Section 15.2, "Passing Parameters to an ADF Bounded Task Flow"](#).

14.6.3 How to Call an ADF Bounded Task Flow Located in Another Web Application

You can use the **remote-app-url** option on a task flow call activity to call an ADF bounded task flow located in a different web application. You specify in **remote-app-url** an EL expression that, when evaluated, returns the remote web application's URL. Using an EL expression allows you to configure the remote application's URL in any manner, including using context initialization parameters in `web.xml`, for example `# {initParam.remoteAppUrl}`.

In addition to the **remote-app-url**, you must also specify a task flow reference in the task flow call activity metadata. The task flow reference and remote application URL values are combined at runtime to generate a URL to the called task flow.

To call an ADF bounded task flow in a different web application:

1. In the Component Palette, drag the **Task Flow Call** activity from the ADF Task Flow dropdown list to the task flow diagram.
2. In the task flow diagram, select the task flow call activity.
3. In the Property Inspector, expand the **Task Flow Reference** section.
4. Enter a reference (**document** and **id**) to the bounded task flow (see [Section 14.6.1, "How to Call an ADF Bounded Task Flow"](#) for more information).
5. In the Property Inspector, click the button next to **remote-app-url**.

6. In the Expression Builder dialog, you can create an EL expression that, when evaluated, furnishes two components in the URL to the called bounded task flow, the server root and the app context. For example, you could specify the EL expression, `# {pageFlowScope.managedbean.URLmethod}`, where `URLmethod` evaluates to the string `http://my.remote.com:80/myapp/faces`.

14.6.4 How to Call a Bounded Task Flow with a URL

When calling by URL, you are responsible for creating the entire URL. The called bounded task flow can be in the same or a different application.

You can use an EL Expression to create the URL. For example, the EL expression could evaluate to a server root of `somedomain/internalApp` and an app context of `MyApp` as shown in [Example 14–9](#).

Example 14–9 Sample URL for an ADF bounded task flow

```
http://somedomain.com/internalApp/MyApp/faces/adf.task-flow?adf.tfid=displayHelp&adf.tfDoc=%2FWEB-INF%2Fdisplayhelp.xml&topic=createPurchaseOrder
```

[Example 14–10](#) contains the syntax for a URL to an ADF bounded task flow.

Example 14–10 URL Syntax For Call to ADF Bounded Task Flow Using Named Parameters

```
<server root>/<app_context>/faces/adf.task-flow?adf.tfid=<task flow definition ID>&adf.tfDoc=<document name>&<named parameter>=<named parameter value>
```

The components of the URL syntax are:

- **<server root>**: Provided by customization at site or admin level, for example, `http://mycompany.com/internalApp`. The root name depends on the server where the task flow definition is deployed. The ADF bounded task flow URL is a resource within the JSF servlet's URL path.
- **<app context>**: The Web application context root, for example, `MyApp`. The context root is the base path of a Web application.
- **faces**: Faces servlet mapping.
- **adf.task-flow**: A reserved word that identifies the ADF Controller for the remote web application.
- **adf.tfid**: A URL parameter that supplies the task flow ID to be called.
- **<task flow ID>**: The identifier of the task flow definition to be called, for example, `displayHelp`. This is the same task flow ID that is used when calling locally. Note that this identifier is not the same as the task flow call activity instance ID. The parameter value must be represented as a string.
- **adf.tfDoc**: A URL parameter that supplies the document name containing the task flow definition ID to be called.
- **<document name>**: A document name containing the task flow definition ID to be called, for example, `%2FWEB-INF%2FtoUppercase%2FdisplayHelp.xml`. If you are handcrafting the ADF bounded task flow URL, you are responsible for the appropriate encoding.
- **<named parameter>**: (optional) The name of an input parameter definition for the called ADF bounded task flow, for example, `topic`. You must supply all required input parameter definitions.

- <named parameter value>: (optional) The value of the input parameter.

Note: URL parameter names that begin with an underscore ('_') are intended for internal use only and should not be used. Although you may see these names on URLs generated by ADF controller, you should not attempt to use or depend on them.

14.6.5 Specifying a Parameter Converter

A parameter converter is an EL value expression that evaluates to an object of type `oracle.adf.controller.URLParameterConverter`. If a converter is specified, it is used to convert task flow parameter values to / from the string representation used in a URL.

14.6.6 How to Specify Before and After Listeners

Task flow call activity before and after listeners are used to identify the start and end of an ADF bounded task flow. Specifying a listener in the task flow call activity means that the listener executes on that specific usage of the called task flow definition.

You specify the listener as an EL expression for a method that will be called upon entry or exit of an ADF bounded task flow, for example, `<before-listener>#{global.showState}</before-listener>`. The method cannot have parameters.

- Before Listener: An EL expression for a Java method called before an ADF bounded task flow is entered. It is used when the caller needs to know when an ADF bounded task flow is being initiated.
- After Listener: An EL expression for a Java method called after an ADF bounded task flow returns. It is used when the caller needs to know when an ADF bounded task flow exits and control flow returns to the caller.

If multiple before listeners or multiple after listeners are specified, they are called in the order in which they appear in the source document for the unbounded or bounded task flow. A task flow call activity can only have one before listener and one after listener.

In order for the task flow call after listeners to be called, control flow must return from the ADF bounded task flow using a control flow rule. If an end user leaves an ADF Bounded task flow using the browser Back button or other URL, task flow call after listeners will not be called. You must use a task flow definition finalizer to release all acquired resources and perform cleanup of an ADF bounded task flow that the end user left by clicking a browser back button. See [Section 17.1, "Using Initializers and Finalizers"](#) for more information.

To specify a before or after listener on a task flow call activity:

1. In the diagram of the calling ADF bounded task flow, select the task flow call activity.
2. In the Property Inspector, click **Listeners**.
3. Click the button next to either **before-listener** or **after-listener**.
4. In the Expression Builder dialog, drill down to the Java class containing the method for the listener.
5. Open the class node and select the listener method.

When you are done, your EL expression might look like
`#{{pageFlowScope.managedBean.methodListener}}`.

6. Click **OK**.

14.6.7 What Happens When You Add a Task Flow Call Activity

After you add a task flow call activity to a task flow diagram, you must specify a reference to the called ADF bounded task flow using one of the methods described in [Section 14.6.1, "How to Call an ADF Bounded Task Flow"](#). For example, if you drop an existing bounded task flow on the task flow call activity, JDeveloper generates the task flow reference automatically. The task flow reference is used to invoke the called ADF bounded task flow. Each task flow reference consists of:

- **id**: The task flow definition id contained in the XML source file for the called ADF bounded task flow. For example, a called task flow might have an id called `targetFlow`. The same XML source file can contain multiple ADF bounded task flow definitions, each definition identified by a unique id.

Note: If you use JDeveloper to create the ADF bounded task flow, there is only one task flow definition per document.

- **document**: The name of the XML source file containing the **id** of the called ADF bounded task flow. If **document** is not specified, `adfc-config.xml` is assumed.

[Example 14-11](#) contains an example task flow reference within a task flow call activity. In order to invoke a bounded task flow, you need to know its id and name of the file containing the id.

Example 14-11 Task Flow Reference

```
<adfc-config xmlns="http://xmlns.oracle.com/adf/Controller">
.
.
.
<task-flow-definition id="task-flow-definition">
  <default-activity>view1</default-activity>
  <task-flow-call id="taskFlowCall">
    <task-flow-reference>
      <document>/WEB-INF/called-task-flow-definition.xml</document>
      <id>called-task-flow-definition</id>
    </task-flow-reference>
  </task-flow-call>
</task-flow-definition>
.
.
.
</adfc-config>
```

14.6.8 What Happens at Runtime: Using a Task Flow Call Activity

The ADF Controller performs the following steps when an ADF bounded task flow is called using a task flow call activity:

1. Verifies that the user is authorized to call the ADF bounded task flow.

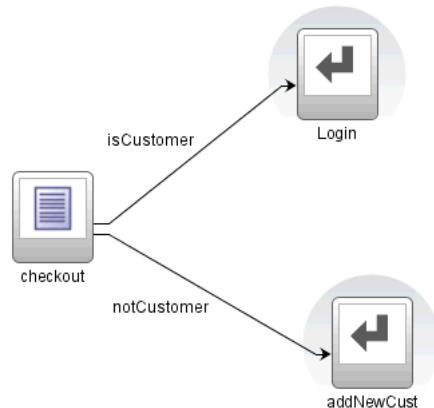
2. Invokes task flow call activity before listener or listeners, if specified (see [Section 14.6.6, "How to Specify Before and After Listeners"](#)).
3. Evaluates the input parameter values on the ADF bounded task flow.
4. Pushes the called ADF bounded task flow onto the stack and initializes its pageFlow scope
5. Sets input parameter values in the called ADF bounded task flow's context
6. Invokes an ADF bounded task flow initializer method, if one is specified (see [Section 17.1, "Using Initializers and Finalizers"](#) for more information).
7. Executes the ADF bounded task flow's default activity.

14.7 Using Task Flow Return Activities

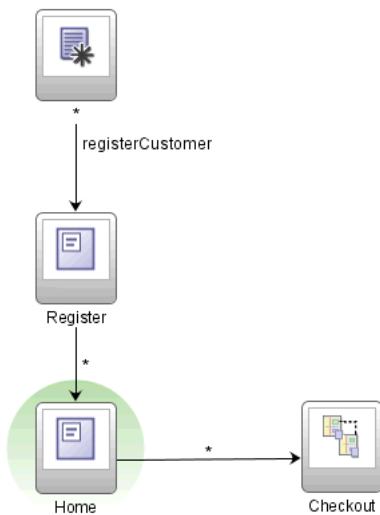
The task flow return activity identifies the point in the application control flow where an ADF bounded task flow completes and sends control flow back to the caller. You can use the task flow return activity only within an ADF bounded task flow.

A gray circle around an activity icon indicates that the activity is an exit point for an ADF bounded task flow. Each ADF bounded task flow can have zero to many task flow return activities. In [Figure 14–8](#), the ADF bounded task flow contains two task flow return activities.

Figure 14–8 Multiple Task Flow Return Activities



Each task flow return activity specifies an **outcome** that is returned to the calling task flow. For example, the **outcome** for the AddNewCust task flow return activity is registerCustomer. As shown in [Figure 14–9](#), the calling task flow can match this outcome with a control flow case **from-outcome**. This handles control flow upon return from the called task flow.

Figure 14–9 Control Flow Case Specified on Calling Task Flow

The **restore-save-point** option specifies whether model changes made within an ADF bounded task flow should be discarded when exiting using a task flow return activity. If set to true, transactions are rolled back to the ADFm save point that was created when the ADF bounded task flow was originally entered. You can specify this option only if the bounded task flow on which the task flow return activity is located was entered without starting a new transaction. For more information, see [Section 17.2.1, "How to Enable Transactions in an ADF Bounded Task Flow"](#)

To add a task flow return activity to an ADF bounded task flow:

1. Drag a task flow return activity from the ADF Task Flow dropdown list in the Component Palette to the diagram for the ADF bounded task flow.
2. In the task flow diagram, select the task flow return activity.
3. On the Common page of the Property Inspector, expand the **Outcome** section.
4. In the **name** field, enter an outcome, for example, preferredCustomer.

Specifying this will return an outcome to the caller when the ADF bounded task flow exits. You can specify only one outcome per task flow return activity. The calling ADF task flow should define control flow rules to handle control flow upon return. See [Section 13.2.3, "How to Add Control Flows"](#) for more information.

5. In Property Inspector, click **Behavior**.
6. In the **Reentry** dropdown list, choose either **reentry-allowed** or **reentry-not-allowed**.
 - **reentry-allowed:** Reentry is allowed on any view activity within the ADF bounded task flow.
 - **reentry-not-allowed:** Reentry of the ADF bounded task flow is not allowed. If you specify **reentry-not-allowed** on a task flow definition, an end user can still click the browser back button and return to a page within the bounded task flow. However, if the user does anything on the page such as clicking a button, an exception (for example, `InvalidTaskFlowReentry`) is thrown indicating the bounded task flow was reentered improperly. The actual reentry condition is identified upon the submit of the reentered page.

Your selection defines the default behavior when the ADF bounded task flow is reentered by an end user clicking a browser back button. This selection applies only if **reentry-outcome-dependent** has been set on the ADF bounded task flow where the task flow return activity is located. For more information, see [Section 17.3, "Reentering an ADF Bounded Task Flow"](#).

7. In the **End Transaction** dropdown, list choose either **commit** or **rollback**.
 - **commit**: commits the existing transaction to the database.
 - **rollback**: rolls back the transaction to what it was on entry of the called task flow. This has the same effect as cancelling the transaction, since it rolls back a new transaction to its initial state when it was started on entry of the bounded task flow.If you do not specify commit or rollback, the transaction is left open to be closed by calling ADF bounded task flow.
8. In the **restore-save-point** dropdown list, select **true** if:
 - if **new-transaction** is not selected on the bounded task flow bounded task flow on which the task flow return activity is located
 - ADFm model changes made within an ADF bounded task flow should be discarded when exiting using the task flow call activity. The transaction is rolled back to the save point created on entry of the ADF bounded task flow.

For more information, see [Section 17.2.1, "How to Enable Transactions in an ADF Bounded Task Flow"](#).

14.8 Using Save Point Restore Activities

The Save Point Restore activity allows you to restore a previous persistent save point in an application supporting save for later functionality. A save point captures a snapshot of the Fusion web application at a specific instance. Save Point Restore enables the application to restore whatever was captured when the save point was originally created.

When a save point is restored, the ADF Controller terminates the saved application and restarts the application that was executing when the end user performed a save. The end user's original location in the application is displayed. Once the save-point-id is restored, it is deleted from its method of persistence (database or Java object cache). See [Section 17.11.1, "Specifying Save for Later Settings"](#) for more information.

A save point restore activity is not required within every individual application supporting Save For Later capabilities. It is only required within the applications responsible for restoring the previously persistent save-point-ids. For example, a save point restore activity would not be required within a Create Expense Report application, but would be within the application used to select previously saved Expense Reports for further updates.

[Section 17.5, "Saving for Later"](#) contains detailed information about enabling save for later capabilities in a task flow and provides an example of how to use the Save Point Restore activity to retrieve the saved application state and data.

14.9 Using Parent Action Activities

An ADF bounded task flow running in an ADF region may need to trigger navigation of its enclosing view. The parent action activity allows an ADF bounded task flow to generate outcomes that are passed to its parent. The outcomes are used to navigate the

ADF task flow containing the enclosing view's rather than navigating the ADF task flow of the ADF region.

- Parent Outcome: Specifies a value passed to the parent viewport to navigate the enclosing view's task flow rather than navigating the region's task flow where the Parent Action activity is defined.
- Outcome: Specifies a control flow outcome within the region after the parent outcome is queued to the parent.

This is useful in cases where the parent doesn't navigate as a result of the parent outcome sent by the region and the region doesn't want to continue displaying the same view. If outcome is not specified, the region's viewId will remain unchanged.

Using Parameters in Task Flows

This chapter describes how to specify parameters in view activities and in ADF bounded task flows. It includes the following sections:

- [Section 15.1, "Passing Parameters to a View Activity"](#)
- [Section 15.2, "Passing Parameters to an ADF Bounded Task Flow"](#)
- [Section 15.3, "Sharing Data Control Instances"](#)
- [Section 15.4, "Specifying Return Values"](#)
- [Section 15.5, "Specifying EL Binding Expressions"](#)

15.1 Passing Parameters to a View Activity

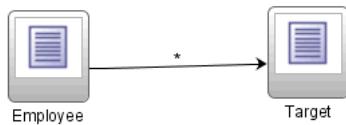
You can use view activity input page parameters as aliases. The alias allows you to map task flow definition input parameters to page parameters. The view activity input page parameters map managed beans and any information available to the calling task flow to a managed bean on the page itself. To pass values out of view activities, values should be stored in pageFlow scope or managed beans. (For information about using view activities in a task flow, see [Section 14.2, "Using View Activities"](#)).

For example, the page might specify `# {pageFlowScope.empno}` as a page parameter and a task flow definition might specify `# {pageFlowScope.employeeID}` as the value of an input parameter definition.

The **from-value** on the view activity input page parameter would be `# {pageFlowScope.employeeID}` and the **to-value** would be `# {pageFlowScope.empno}`. This enables reuse of both the page definition and task flow definition because you don't have to redefine parameters for every context in which each is used.

Other values contained within the task flow can be mapped to input page parameters, not just task flow definition input parameter definition values.

The example in the steps below describes how to specify an input page parameter mapping in the Employee view shown in [Figure 15-1](#). Once you set this up, you can pass a parameter to the Employee activity as a pageFlowScope value or a value on a managed bean. The Employee activity can pass a value to the Target activity by placing it within pageFlowScope or a managed bean to the Target activity, based on the EL expression you specified in the **to-value**.

Figure 15–1 Task Flow with Two Activities**To set an input page parameter value and retrieve it:**

1. In the editor, open the task flow diagram.
2. In the task flow diagram, select the view activity.
3. In the Property Inspector, click **Page Parameters**.
4. In the **Input Page Parameter** section, click the + icon.
5. Enter a **from-value** using an EL expression, that when evaluated, that specifies where the input page parameter value will be passed from, for example, # {pageFlowScope . EmployeeID}.

The task flow shown in [Figure 15–1](#) can set an input parameter definition value that the view activity can retrieve. To retrieve the parameter value, you can specify a **from-value** on the view activity that matches the **Value** specified for the ADF bounded task flow input parameter definition.

6. Enter a **to-value** using an EL expression that, when evaluated, specifies where the page associated with the view activity can retrieve the value of the input page parameter, for example, # {pageFlowScope . EmployeeNo}.

The **to-value** is the second part of the view activity input page parameter. It specifies where the page associated with the view can retrieve the value of the view parameter.

15.2 Passing Parameters to an ADF Bounded Task Flow

A called ADF bounded task flow can accept input parameters and can pass return values to the caller upon exit (see [Section 15.4, "Specifying Return Values"](#) for more information).

Note: Instead of explicitly passing a data controls as parameters between task flows, you can simply share them by specifying the **data-control-scope** option on the called bounded task flow. For more information, see [Section 15.3, "Sharing Data Control Instances"](#).

To pass an input parameter to a bounded task flow, you must specify one or more:

- Input parameters on the task flow call activity. These specify where the calling task flow will store parameter values.
- Input parameter definitions on the called bounded task flow. These specify where the called bounded task flow can retrieve parameter values.

If you call a bounded task flow using a URL rather than a task flow call activity, you pass parameters and values on the URL itself. See [Section 14.6.3, "How to Call an ADF Bounded Task Flow Located in Another Web Application"](#) for more information.

The input parameter **name** specified for each of the above options will be the same in order to map input parameter values back into the called bounded task flow. The **value** for each corresponds to the mapping between where the value will be retrieved within the caller and the called task flow.

If you don't specify a value for the input parameter, the value defaults to `# {pageFlowScope.parmname}`, where `parmname` is the name of your parameter.

You can specify on the input parameter definition for the called bounded task flow whether an input parameter is required. If a required input parameter is not received, an error occurs (these are flagged at design time as well as runtime). An input parameter definition that is identified as not required can be ignored during task flow call activity creation.

By default, all objects are passed by reference. Task flow call activity input parameters can be passed by reference only if managed bean objects are passed, not individual values. By default, primitive types (for example, int, long, or boolean) are passed by value.

The **pass by value** checkbox only applies to objects, not primitives and is used to override the default setting of passing by reference. Mixing the two, however, can lead to unexpected behavior in cases where parameters reference each other. If input parameter A on the task flow call activity is passed by value and input parameter B on task flow call activity is passed by reference, and B has a reference to A, the result can be two different instances of A and B.A).

[Example 15–1](#) shows an input parameter definition specified on a task flow definition for a bounded task flow.

Example 15–1 Input Parameter Definition

```
<task-flow-definition id="sourceTaskflow">
    .
    .
    .
    <input-parameter-definition>
        <name>inputParameter1</name>
        <value>#{pageFlowScope.parmValue1}</value>
        <class>java.lang.String</class>
    </input-parameter-definition>
    .
    .
    .
</task-flow-definition>
```

[Example 15–2](#) shows the input parameter metadata that would be specified on the task flow call activity that called the bounded task flow shown in [Example 15–1](#).

Example 15–2 Input Parameter on Task Flow Call Activity

```
<task-flow-call id="taskFlowCall1">
    .
    .
    .
    <input-parameter>
        <name>inputParameter1</name>
        <value>#{pageFlowScope.newCustomer}</value>
        <pass-by-value/>
    </input-parameter>
    .

```

```
</task-flow-call>
```

Best Practice:

You can set parameters for a page using an ADF task flow as opposed to dropping a parameterized form from the data control panel or using a form with a method which takes parameters that are invoked using an invoke actions.

The first technique is a way of passing parameters to a page and the others a way of consuming parameters on a page. If a page needs parameters, you should pass them using task flow parameters or by setting scope variables.

The following steps describe passing an input parameter from a source task flow to a target bounded task flow using a task flow call activity. Although you can pass parameter values from any activity on the source task flow, the passed parameter in the steps below will contain the value of an input text field on a page in the source task flow.

Before you begin, create a calling and called task flow (see [Section 13.2, "Creating Task Flows"](#) for more information). The caller can be a bounded or unbounded task flow. The called task flow must be a bounded task flow. On the calling task flow, add the activities shown in [Figure 15–2](#). The page associated with the view on the calling task flow should contain an input text field and a button. When an end user clicks the button on the JSF page, control should pass to the task flow call activity.

Figure 15–2 Calling Task Flow



To pass an input parameter to an ADF bounded task flow:

1. Select the input text component on the JSF page.
2. In the Property Inspector, enter a **value** for the input text component.
For example, you can specify the value as an EL expression, for example
`# {pageFlowScope.inputValue}`.
3. In the Application Navigator, double-click the name of the called task flow to open its diagram.
4. Click the **Overview** tab for the called task flow.
5. Click **Parameters** and expand the **Input Parameter Definition** node.
6. Click the + icon next to **Input Parameter Definition**.
7. In the **Name** field, enter a name for the parameter, for example, `inputParm1`.

8. In the **Value** field, enter an EL expression where the parameter value is stored and referenced, for example, `# {pageFlowScope.inputValue}`.
9. In the **Class** field, enter a Java class for the input parameter definition. If you leave the Class field empty, its value is by default implicitly set to `java.lang.String`.

Best Practice:

It is a best practice to always explicitly specify a value in the **Class** field even when the value is `java.lang.String`.

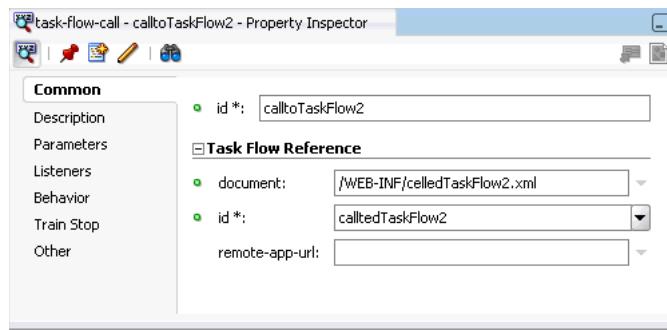
This ensures that a value or object of the correct type is passed to the input parameter. If the task flow definition is implemented using a task flow call activity, as an ADF region or as an ADF dynamic region, input parameter values accept `# {bindings.bindingId.inputValue}` binding EL expression syntax (see [Section 15.5, "Specifying EL Binding Expressions"](#) for more information).

If **Class** is implicitly set to `java.lang.String` and the `# {bindings.bindingId}` EL expression syntax is used, the parameter values that are passed are the actual binding objects instead of the values of the bindings. As a result, the bindings are evaluated within the wrong context, the context of the task flow call, ADF region, or ADF dynamic region, instead of the context of the page. Multiple data control instances are created, eventually setting the wrong data control context in the frame.

10. In the editor, open the diagram for the calling task flow.
11. In the Application Navigator, drag the called ADF bounded task flow and drop it on top of the task flow call activity that is located on the calling task flow.

Dropping a bounded task flow on top of a task flow call activity in a diagram automatically creates a task flow reference to the bounded task flow. As shown in [Figure 15–3](#), the task flow reference contains the bounded task flow id and a document name. The document name points to the XML source file that contains the id.

Figure 15–3 Task Flow Reference in Property Inspector



12. In the Property Inspector for the task flow call activity, click **Parameters** and expand the **Input Parameters** section.
13. Enter a **name** that identifies the input parameter.

Because you dropped the bounded task flow on a task flow call activity having defined input parameters, the **name** should be already be specified. You must keep the same input parameter name.

14. Enter a parameter **value**, for example, # {pageFlowScope.parm1}.

The value on the task flow call activity input parameter specifies where the calling task flow will store parameter values.

The value on the input parameter definition for the called task flow specifies where the value will be retrieved for use within the called task flow definition once it is passed.

15. At runtime, the called task flow is able to use the input parameter. Since you specified pageFlowScope on the **value** in the input parameter definition for the called task flow, you can use the parameter value anywhere in the called ADF bounded task flow. For example, you can pass it to a view activity on the called bounded task flow. See [Section 14.2.2.2, "What Happens When You Transition Between Activities"](#) for more information.

15.3 Sharing Data Control Instances

You can share data control instances between task flows (for more information about data controls, see [Section 11.2, "Exposing Application Modules with Oracle ADF Data Controls"](#)). A called bounded task flow can reference and modify the value of the data control owned by its calling task flow. This allows the called task flow to share the same data control instance as its parent. Both task flows look in the same place, the data control frame, to get the data control instance.

In the Fusion Order Demo for example, you may have a value that is used to display a row in a product table on an ADF page. Clicking a button on the page updates a shopping cart form in an ADF region. The form updates with the product name, based on the selected value in the table.

A *data control frame* is the container associated with a bounded task flow that contains data control instances. The bounded task flow's transactions operate on data control instances in the data control frame. A new `DataControlFrame` is created for each task flow that is entered. This results in each ADF task flow having its own unique instance of any data control it uses.

To specify whether data control instances are shared between the calling and called task flows, you must set a **data-control-scope** value of either **shared** or **isolated** on the called bounded task flow. **shared** is the default value.

If a **shared** data-control-scope is specified on both the calling and called task flow definition, the data control frame used will be the one from whoever called the calling task flow. In [Example 15-3](#), the data control frame for Task Flow Definition A would be also used by both B and C.

If an **isolated** data-control-scope is specified on both the calling and called task flow definition, they both will use their own data control frames.

Example 15-3

```
Task Flow Definition A - isolated
  Task Flow Definition B - shared
    Task Flow Definition C - shared
```

Note: An ADF bounded task flow with a **shared** data-control-scope that is called using a URL cannot share data control instances with the calling task flow. If the called bounded task flow specifies **shared** as the **data control-scope**, ADF Controller will throw an exception.

A caller of a bounded task flow can share its caller's data control instances to any depth. For example, task flow A can share data control instances with called bounded task flow B. Called bounded task flow C can share the same data control instances.

An ADF bounded task flow specifying a shared data control scope used within an ADF region shares the data control instances of the task flow in the parent ViewPort (for more information, see [Section 16.1.10, "What You May Need to Know about ViewPorts"](#)).

A new data control frame is created for the ADF unbounded task flow in each RootViewPort's call stack. Therefore, each browser window is isolated from all other browser windows within the same HTTP session. The browser windows do not share data control instances. However, if an end user uses the CTRL+N keys to open a new browser window, the two windows have the same server-side state and share data control instances.

For performance reasons, data controls are not instantiated until needed.

Before you begin the following steps, create a calling and called task flow.

To share a data control between task flows:

1. In the Property Inspector for the called task flow, select **Behavior**.
2. In the **data-control-scope** list, choose **shared**.

The called task flow will share all data control instances with its caller.

Note: After you set the **data-control-scope**, you may also need to check the **transaction** option for the bounded task flow to determine if there are any interactions between the options. See [Section 15.3.1, "What You May Need to Know About Managing Transactions"](#) for more information.

3. The **data-control-scope** on the calling task flow definition should be set to **isolated** if you don't want it to be dependent on any task flow definition above it. By default, the **data-control-scope** for all task flow definitions is **isolated**.

15.3.1 What You May Need to Know About Managing Transactions

Data control instances cannot be shared across more than one transaction at the same time. If your task flow is involved in managing transactions, the value you select for the **data-control-scope** option may affect the **transaction** option settings for a bounded task flow.

Table 15–1 Interaction of transaction and data-control-scope option settings

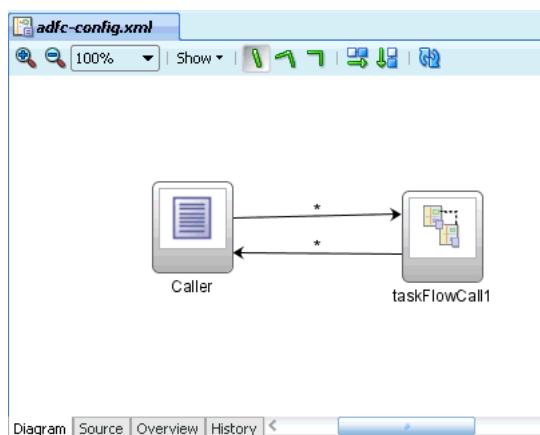
transaction Option	data-control-scope Isolated Option	data-control-scope shared Option
requires-existing-transaction	Invalid.	If a transaction is not already open, an exception is thrown. The existing transaction is never committed.
requires-transaction	Always begins a new transaction	Begins a new transaction if one is not already open.
new-transaction	Always begins a new transaction	Begins a new transaction if one is not already open. If one is already open, an exception is thrown.
<default> (none)	A new DataControlFrame is created without an open transaction.	NA

15.4 Specifying Return Values

As shown in [Figure 15–4](#), a task flow return activity causes the ADF bounded task flow to return to the task flow that called it. The called bounded task flow can pass return values back to the calling task flow.

Figure 15–4 ADF Bounded Task Flow Returning to a Calling Task Flow

The values are returned to the calling task flow (see [Figure 15–5](#)).

Figure 15–5 ADF Unbounded Task Flow Calling a Bounded Task

To return a value, you must specify:

- Return value definitions on the called bounded task flow. These specify where the return value is to be taken from upon exit of the called bounded task flow.
- Return values on the task flow call activity in the calling task flow. These specify where the calling task flow can find return values

The caller of the bounded task flow can choose to ignore return value definition values by not identifying any task flow call activity return values back to the caller.

Return values on the task flow call activity are passed back by reference. Nothing inside the ADF bounded task flow will still reference them, so there is no need to pass by value and make a copy.

Before beginning the following steps, create a calling task flow (can be either bounded or unbounded) and a target bounded task flow.

To specify a return value for a called ADF bounded task flow:

1. In the task flow editor, open the ADF bounded task flow that will be called.
2. Click the **Overview** tab.
3. Click **Parameters** and expand the **Return Value Definitions** section.
4. Click the + icon next to Return Value Definitions.
5. In the **name** field, enter a name to identify the return value, for example, `Return1`.
6. In the **value** field, enter an EL expression that specifies where the return value is to be taken from upon exit of the called bounded task flow, for example, `# {pageFlowScope.ReturnValueDefinition}`.
7. In the task flow editor, open the calling ADF task flow.
8. In the Component Palette, drag the task flow call activity from the ADF Task Flow list and drop it on the diagram for the ADF calling task flow.
9. In the task flow diagram, select the task flow call activity.
10. In the Property Inspector, click **Parameters** and expand the **Return Values** section.
11. In the **name** field, enter a name to identify the return value, for example, `Return1`.

The name of the return value must match the name of the return value definition on the called task flow definition.

Tip: If you drop the bounded task flow that you intend to call on the task flow call activity, the **name** field will already be specified.

Therefore, the **name** field for the return value will automatically match the **name** of the return value definition on the called task flow definition.

12. In the **value** field, enter an EL expression that specifies where the calling task flow can find return values, for example, `# {pageFlowScope.ReturnValue}`.

15.5 Specifying EL Binding Expressions

If a task flow definition is implemented using a task flow call activity, as an ADF region or as an ADF dynamic region, you can specify parameter values using standard EL expression syntax. For example, you can specify parameters using

`#{bindings.bindingId.inputValue}` or `#{bindings.bindingId}` or simply
`#{inputValue}` syntax.

[Example 15–4](#) shows an example of a taskFlow binding for an ADF region.

Example 15–4 Example ADF Region taskFlow Binding

```
<taskFlow id="Department1" taskFlowId="/WEB-INF/Department.xml#Department"
          xmlns="http://xmlns.oracle.com/adf/Controller/binding"
          Refresh="ifNeeded">
    <parameters>
        <parameter id="DepartmentId" value="#{bindings.DepartmentId.inputValue}"
                   xmlns="http://xmlns.oracle.com/adfm/uimodel"/>
    </parameters>
</taskFlow>
```

Appending `inputValue` to the parameter value binding EL expression, ensures that the parameter is assigned the value of the binding rather than the actual binding object.

If you use the above syntax without appending `inputValue` `#{bindings.bindingId}`, the binding object (not the value of the binding) is passed. Therefore, the binding will end up being evaluated within the wrong context (the context of the Task Flow Call, ADF Region, or ADF Dynamic Region instead of the context of the page) and multiple Data Control instances will be created eventually setting the wrong Data Control context in the frame.

16

Using ADF Task Flows as Regions

This chapter includes the following sections:

- [Section 16.1, "Introduction to ADF Regions"](#)
- [Section 16.2, "Creating ADF Dynamic Regions"](#)
- [Section 16.3, "Creating Dynamic Region Links"](#)

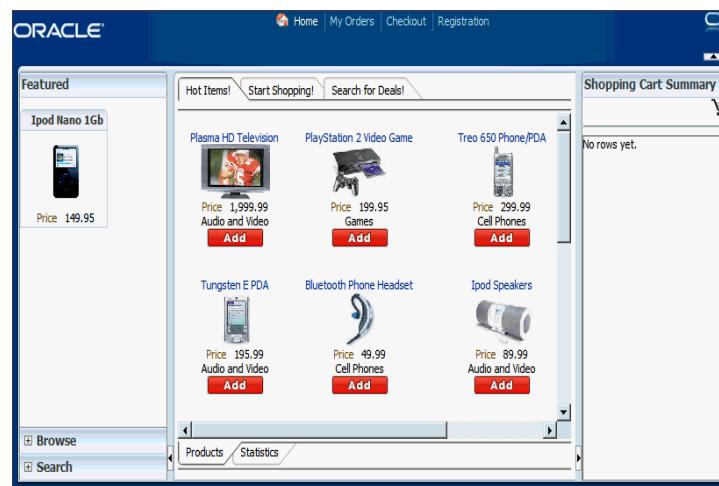
16.1 Introduction to ADF Regions

You can add an ADF bounded task flow to a page as an ADF region. A region is a UI component whose content is based on a task flow definition. When first rendered, the region's content is that of a first view activity in task flow.

Any view activity used in the region must be created using page fragments, not pages. A page fragment is a JSF JSP document that can be rendered as content of another JSF page. It is similar to a page, except it cannot contain elements such as `<f:view>` (see [Section 16.1.9, "What You May Need to Know About Page Fragments"](#) for more information).

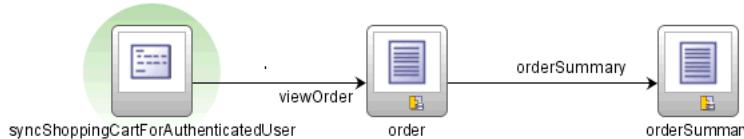
An ADF region appears as a sub-area on a page. The Shopping Cart Summary shown in [Figure 16–1](#) is an ADF bounded task flow exposed as an ADF region on the Fusion Order Demonstration application Home page. The task flow contains a single view activity summarizing the contents of the cart. The region drives the presentation of the Shopping Cart ADF bounded task flow as it executes on the page.

Figure 16–1 Shopping Cart Summary in Fusion Order Demonstration Home Page



In a more complicated region, the ADF bounded task flow can be comprised of a series of separate activities. The user might view each page fragment contained in the ADF bounded task flow one after another. To navigate between page fragments, the user clicks UI controls located on each page fragment. As shown in [Figure 16–2](#), the Checkout bounded task flow contains method call and view activities. The order view activity is a page fragment that displays an order before the end user clicks a button to transition to a page fragment that displays a summary.

Figure 16–2 Checkout Bounded Task Flow

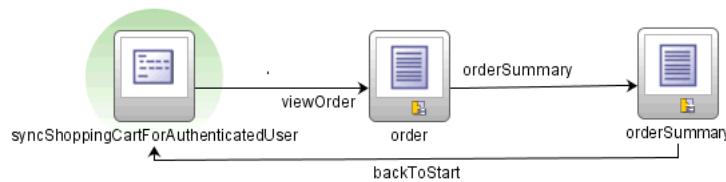


A primary reason for executing an ADF task flow as an ADF region is reuse. You can isolate a small, specific piece of application functionality as a region that can be reused throughout the application. ADF regions enable you to extract, parameterize, and package this reusable content within a bounded task flow so that it can be added to other pages. They can be reused wherever needed, which means they are not dependent on a parent page. If you modify an ADF bounded task flow definition, the changes are applied to any ADF region that uses the task flow.

The Fusion Order Demonstration application contains several functional areas that are candidates for reuse. The Shopping Cart Summary shown in [Figure 16–1](#) might be used as a region on the Home page so that the end user can view any items in the cart saved from a previous session or on a Browse results page in order to add the currently displayed item to the cart. You could pass different parameters to each bounded task flow instance to determine the list of products shown in each ADF region.

When you create the bounded task flow you plan to use as a region on one or more pages or page fragments, determine the pieces of information required by a task. You can define these declaratively as input parameter definitions on the bounded task flow and pass their values into the region. When you are finished, you can drop the bounded task flow as region on one or more pages.

All of the ADF task flow activities that appear in the Component Palette can be added to the ADF bounded task flow that will be used as a region. However, adding a task flow return activity to an ADF region is not recommended because there is no caller to return to except the page or page fragment on which the region is displayed. Instead, control flow can be designed so that it recycles back to the beginning of the task flow. As shown in [Figure 16–3](#), you can add a control flow case that leads from the last activity that executes to an activity towards the beginning of the task flow. For more information see [Section 16.1.8, "What You May Need to Know About Returning from an ADF Region"](#).

Figure 16–3 Checkout Bounded Task Flow Recycling to Beginning

16.1.1 How to Create an ADF Region

To create an ADF region, you must create an ADF bounded task flow that contains at least one view activity or one task flow call activity. Otherwise, the ADF region may not have anything to display. View activities within the region must be associated with page fragments, not pages.

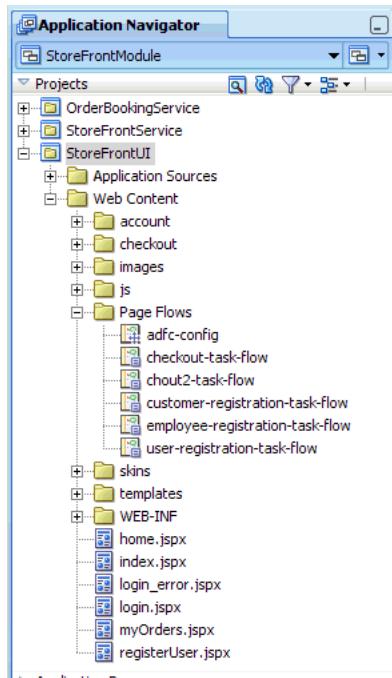
Before you begin the steps described below, create the following:

- ADF bounded task flow containing page fragments (see [Section 13.2, "Creating Task Flows"](#) for more information).
- Input parameter definition for the ADF bounded task flow (see [Section 15.2, "Passing Parameters to an ADF Bounded Task Flow"](#) for more information).
- JSF page onto which you will drop the ADF bounded task flow as a region (see the *Oracle Fusion Middleware Web User Interface Developer's Guide for Oracle Application Development Framework* for more information).

To add an ADF bounded task flow as a region to a JSF page:

1. In the Application Navigator, drag the ADF bounded task flow onto the page and drop it where you want the region to be located.

You can often find task flows in the Application Navigator under the Page Flows or WEB-INF folders, as shown in [Figure 16–4](#).

Figure 16–4 ADF Bounded Task Flows in Application Navigator

A menu with choices for adding the task flow as a region or dynamic region displays. In a dynamic region, the ADF bounded task flow that executes within the region is determined at run time. For more information, see [Section 16.2, "Creating ADF Dynamic Regions"](#). If your ADF bounded task flow is not eligible to be dropped as an ADF region (for example, it contains pages, not page fragments), you will not see this menu.

Tip: If you want to use an ADF bounded task flow from a different application than the one in which you are currently working, you can search for it in the Resource Palette. The Resource Palette lets you browse the contents of a catalog and any repositories that are linked into it. The catalog may contain files, services, and components, including ADF bounded task flows, that are of use in the application development process.

For more information about browsing for task flows using the Resource Palette, see [Section 31.2, "Packaging a Reusable ADF Component into an ADF Library"](#).

2. Click **Create > Region**.

Tip: The ADF bounded task flow that you drop must contain only page fragments and no pages. If it contains pages, you will not see the **Create > Region** menu choice when you drop the task flow on the page.

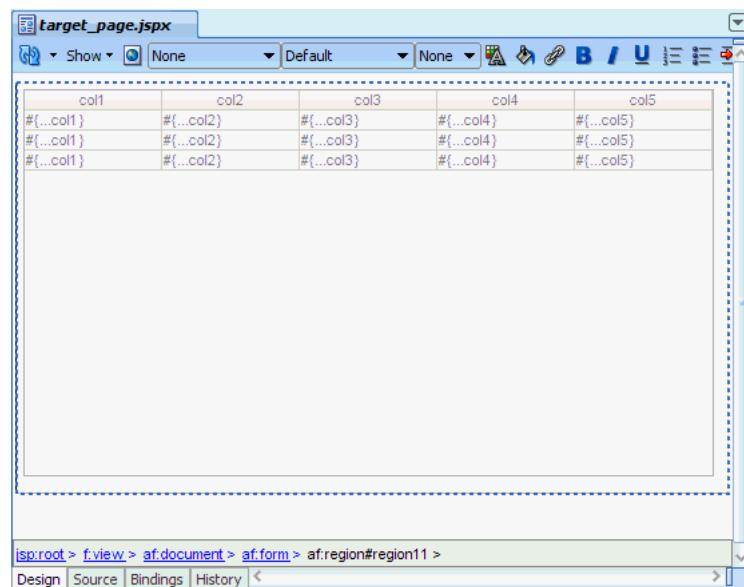
See [Section 13.3, "Adding a Bounded Task Flow to a Page"](#) for more information about adding a ADF bounded task flow that contains pages to another JSF page.

You can convert an ADF bounded task flow that contains pages to one that contains page fragments. See [Section 13.7.3, "How to Convert ADF Bounded Task Flows"](#) for more information.

3. In the editor, select the region you just added to the JSF page.

The region displays inside a dotted boundary, as shown in [Figure 16–5](#).

Figure 16–5 ADF Region in JSF Page



4. JDeveloper automatically populates with default values the following fields in the Property Inspector for the region:
 - **Id:** An id that the JSF page uses to reference the ADF region, for example, `region1`. By default, the id includes the name of the task flow definition for the region.
 - **Value:** An EL reference to the region model, for example, `# {bindings.region1.regionModel}`. This is the region model that describes the behavior of the region.
 - **Rendered:** If true, the region is rendered when the page is rendered.
5. To map parameters between the view activity associated with the JSF page and the ADF region, see [Section 16.1.3, "How to Specify Parameters for an ADF Region"](#).

16.1.2 How to Add a Task Flow Binding Declaratively

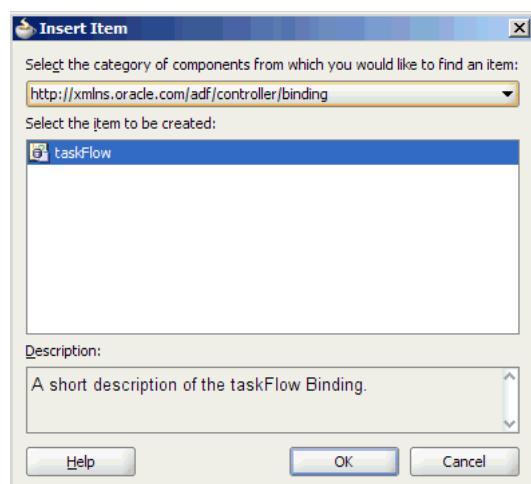
A task flow binding contains input parameter definitions for an ADF bounded task flow that is running inside a region. If you drop an ADF bounded task flow onto a JSF page as a region, JSF automatically creates a task flow binding for the region. You can also add the binding directly to the page definition.

To create a new task flow binding in the page definition:

1. Right-click anywhere on the page and choose **Go to Page Definition**.
2. Expand the **Executables** section
3. Click the plus (+) icon next to **Executables**.

The Insert Item Dialog displays.

Figure 16–6 Insert Item Dialog



4. In the Insert Item dialog, choose.../adf/controller/binding from the drop-down list at the top of the dialog.
5. Click **OK**.

A dialog displays where you can enter the **id** and **taskflow-id**.

16.1.3 How to Specify Parameters for an ADF Region

If you have defined input parameter definitions for the ADF bounded task flow you are exposing as a ADF region or dynamic region, you can map them to the page definition for the JSF page where you are adding the ADF region. This allows the ADF region to reference any values available to the JSF page.

Before you begin, specify an input parameter definition for the ADF bounded task flow you are adding as an ADF region following the steps in [Section 15.2, "Passing Parameters to an ADF Bounded Task Flow"](#).

Best practice:

A page or an ADF region within a page may require information from somewhere else on the page. There are two options for exchanging this information:

- Use input parameters in the ADF task flow
- Use contextual events (see [Section 26.5, "Creating Contextual Events"](#) for more information)

One of the main factors in choosing one of the above options is the nature of the information being shared:

- Use input parameters if the required information is at the very beginning of the ADF task flow and a change of this information will require a restart of the ADF task flow.

For example, you may have a page that contains a table of employees, and an ADF region on that page that contains an ADF task flow for enrolling a selected employee in benefits. A change in the selected employee would require that you restart the benefits enrollment from the beginning for the newly selected employee. Using input parameters to the ADF task flow is the right decision for this case.

You can pass input parameters by reference or by value. If you pass by reference, an update on the main page for the selected employee's information, such as Last Name, is automatically reflected in the ADF task flow running in the ADF region without restarting the ADF task flow.

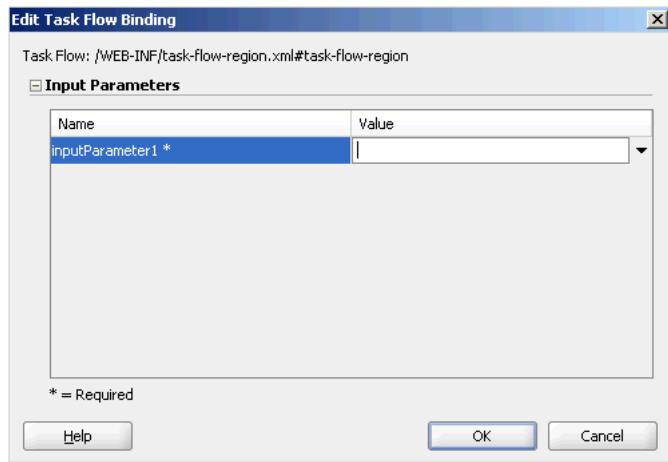
- Use contextual events if the information being exchanged occurs after the start of an ADF task flow or flows, and the change in the information shouldn't restart the ADF task flow.

For example, you may have a page that contains an ADF region showing 360 degrees of a project. The page contains a tab for project setup, a tab for project customers, and a tab displaying team members. The page contains another ADF region on the page that provides a tree for the user to navigate to the various degrees of the project. If the user selects a node in the tree for team members, the region showing the 360 degrees of a project should display the **Team Members** tab. The distinction is that the 360 degree task flow wasn't restarted with a new project. It simply responded to the event by displaying a different view in the ADF task flow. Using contextual events is the right decision for this case.

To map region input parameters to the page:

1. Add the ADF region to the page following the steps in [Section 16.1.1, "How to Create an ADF Region"](#)

If input parameter definitions have been specified on the ADF bounded task flow added as a region, the Edit Task Flow Binding dialog displays.

Figure 16–7 Edit Task Flow Binding Dialog

2. Expand the Input Parameters section.

The **Name** list displays all input parameter definitions that have been specified for the ADF bound task flow. An asterisk (*) indicates that the parameter is required.

- 3.** For each required parameter, specify a **Value**, for example, # {pageFlowScope . inputParameter1}.
- 4.** If the parameter is not required, you can choose to map it to the page or ignore it. If you map it, the parameter is used. To map the parameter definition to the page, specify a **Value**.
- 5.** Click **OK**.

16.1.4 What Happens When You Create an ADF Region

When you drop an ADF bounded task flow onto a page to create an ADF region, a task flow binding is automatically added to the page definition file. As shown in [Example 16–1](#), the task flow binding contains a taskFlowId that consists of a unique task flow identifier (for example, region-task-flow-definition) and the name and path to the XML source file containing the id (for example, /WEB-INF/region-task-flow-definition.xml). If you drop the task flow onto the page as an ADF region, the task flow reference contains a static literal value for the taskFlowId.

Example 16–1 task flow binding for ADF Region

```
<taskFlow id="regiontaskflowdefinition1"
  taskFlowId="/WEB-INF/region-task-flow-definition.xml#region-task-flow-
definition"
  xmlns="http://xmlns.oracle.com/adf/controller/binding"/>
```

taskFlowId can also contain an EL expression that evaluates to a String or to the public class TaskFlowId (`oracle.adf.Controller.TaskFlowId`). See [Section 16.2, "Creating ADF Dynamic Regions"](#) for more information.

An ADF region is comprised of the following:

- The task flow binding
- An ADF bounded task flow

- An ADF region tag (`af:region`)
- A UI component called Region

A task flow binding is added to the page definition each time you add a region to the JSF page. The task flow binding is the bridge between the region UI component and the ADF bounded task flow. It binds a specific region instance to its associated task flow, and maintains all information specific to the ADF bounded task flow.

When you drop the an ADF bounded task flow onto a JSF page, all of the data bindings in the task flow are preserved. A request made for a JSF page containing an ADF region is initially handled like any other JSF page. As the JSF page definition executes, data is loaded to the JSF page. The page contains a `<region>` tag. The page definition contains a task flow binding within the `<executables>` section.

The task flow binding creates an ADF Controller ViewPortContext for its task flow and then asks the ViewPortContext for its current view activity. A viewPort is a display area that has its own `viewId` and is capable of navigating independently of other ViewPorts. A browser window and a region are examples of ViewPorts. The ViewPortContext determines the initial View Activity by executing its task flow until a view activity is reached. The binding container for the current view activity's page and any nested binding containers in this page are then executed.

16.1.5 What Happens at Runtime: Executing an ADF Region

ADF bounded task flows within regions are executed when the page is first created, regardless of the disclosure state of the region UI component. In order to have the ADF task flow executed when the region is disclosed you must trigger a refresh of the task flow binding when the region is disclosed (see [Section 16.1.7, "What You May Need to Know About Refreshing an ADF Region"](#)for more information). If the ADF bounded task flow relies on input parameter values that are not available when the page is first created, you must ensure your task flow will behave correctly when the input parameter values are null.

16.1.6 How to Trigger Navigation of an ADF Region's Parent Task Flow

An ADF bounded task flow running in an ADF region may need to trigger navigation of its enclosing view. JDeveloper provides a way for an ADF task flow to generate an outcome that is passed to its parent. The outcome is then used to navigate the enclosing view's ADF task flow rather than navigating the ADF region's bounded task flow.

For example, you might have a page that displays employee information and an ADF region that contains an enroll button for the employee. After the enroll page is completed and the ADF region returns, then the page refreshes with the next employee.

To trigger navigation of a parent task flow:

1. In the Application Navigator, double-click the XML file for the ADF bounded ask flow that will run in the ADF region.
2. In the ADF Task Flow page of the Component Palette, select **Parent Action** and drag it to the ADF bounded task flow.
3. Section the parent action activity in the ADF bounded task flow diagram.
4. In the Property Inspector, enter a **parent-outcome**, for example, `myregion-parent-outcome`. This value will be sent to the parent and used to

navigate the enclosing view's task flow rather than navigating the ADF region's task flow.

5. In the **outcome** field, enter an optional element that specifies an outcome for control flow within the ADF region after the parent outcome is queued to the parent ADF task flow.

Specifying an **outcome** element is useful in cases where the parent doesn't navigate as a result of the **parent-outcome** that was sent by the region. In addition, the ADF region shouldn't continue to display the same view. If **outcome** is not specified, the region's `viewId` will remain unchanged.

You can specify only literal values for **parent-outcome** and **outcome**.

16.1.7 What You May Need to Know About Refreshing an ADF Region

An ADF region initially refreshes when the parent JSF page on which the region is located first displays. To execute the ADF task flow at the same time that the region renders, you must trigger a task flow binding refresh as the same time that the region renders.

During the initial refresh, any ADF region task flow binding parameter values are passed in from the parent page. The parameter values are used to display the initial page fragment within the ADF Region. If the ADF bounded task flow relies on input parameter values that are not available when the page is first created, ensure your task flow will behave correctly if the input parameter values are null.

An ADF region task flow binding can be refreshed again based on one the following task flow binding attributes:

- **Neither Refresh or RefreshCondition attributes specified (default)**

If neither the Refresh or RefreshCondition task flow binding attributes are specified, the ADF Region is only refreshed once at the time the parent page is first displayed.

- **RefreshCondition="#{EL.expression}"**

The ADF region is refreshed a single time when its RefreshCondition evaluates true. The RefreshCondition must evaluate to a boolean value.

At the time the RefreshCondition is evaluated, if the variable `bindings` is used within the EL Expression, the context refers to the binding container of the parent page, not the page fragment displayed within the ADF Region.

RefreshCondition is independent of the change of value of binding parameters. If the task flow binding parameters do not change, nothing within the ADF region will change.

- **Refresh="ifNeeded"**

Any change to a task flow binding parameter values causes the ADF region to refresh.

There are no values other than ifNeeded for the Refresh attribute.

If the ADF region task flow binding doesn't have parameters, `Refresh="ifNeeded"` is equivalent to not specifying the Refresh attribute.

If you set Refresh to `ifNeeded`, the RefreshCondition attribute should not be specified.

`Refresh="ifNeeded"` is not supported when passing parameters to the task flow binding using a dynamic parameter map. You must instead use the `RefreshCondition="#{EL.Expression}"`.

RefreshCondition and Refresh are mutually exclusive. Refresh="ifNeeded" takes precedence over RefreshCondition. If the bindings variable bindings is used within the EL expression at the time RefreshCondition is evaluated, the context refers to the binding container of the parent page, not the page fragment displayed within the ADF region. The expression is evaluated during the PrepareRender phase of the ADF page lifecycle (for more information, see [Chapter 19, "Understanding the Fusion Page Lifecycle"](#)).

[Example 16–2](#) contains a sample task flow binding located within the page definition of a page on which an ADF region has been added.

Example 16–2 Refresh Option Specified in ADF region binding

```
<taskFlow id="Department1" taskFlowId="/WEB-INF/Department#Department"
          Refresh="ifNeeded"
          xmlns="http://xmlns.oracle.com/adf/controller/binding">
    <parameters>
        <parameter id="DepartmentId" value="#{bindings.DepartmentId.inputValue}"
                   xmlns="http://xmlns.oracle.com/adfm/uimodel"/>
    </parameters>
</taskFlow>
```

If the variable, `bindings`, is used within the EL expression, the context refers to the binding container of the parent page, not the page fragment displayed within the ADF region.

You do not need to refresh an ADF region to refresh the data controls inside the ADF region. During the ADF lifecycle, the refresh events telling the iterators to update will be propagated to the binding container of the current page of the ADF region.

16.1.8 What You May Need to Know About Returning from an ADF Region

ADF bounded task flows within ADF regions are called directly. There is no calling ADF unbounded or bounded task flow to which control can return upon completion. If an ADF bounded task flow is used as a region, it should not exit via normal control flow. Therefore, do not include a task flow return activity within the ADF bounded task flow.

16.1.9 What You May Need to Know About Page Fragments

A page fragment is a JSF JSP document that is rendered as content in another JSF page. In order to act as a fragment, the document source must not contain any of the following tags: `<f:view>`, `<f:form>`, `<html>`, `<head>`, `<body>`. This is because the tags may occur only once in a document or do not support nesting in a JSF JSP page. For example, a page fragment embedded in a page cannot have an `<html>` tag because the JSF JSP page already has one. [Example 16–3](#) contains an example of a simple page fragment. Unlike a JSF JSP page, it contains no `<f:view>` or `<f:form>` tags.

Example 16–3 Page Fragment Source Code

```
<?xml version='1.0' encoding='windows-1252'?>
<jsp:root xmlns:jsp="http://java.sun.com/JSP/Page" version="2.0"
           xmlns:af="http://xmlns.oracle.com/adf/faces/rich">
    <af:commandButton text="Go"/>
    <af:inputText/>
```

```
</jsp:root>
```

The file extension for a page fragment is `.jsff`. An ADF bounded task flow that is added to a JSF JSP page as an ADF region must use page fragments, not pages (see [Section 16.1, "Introduction to ADF Regions"](#) for more information).

16.1.10 What You May Need to Know about ViewPorts

A ViewPort is a display area that has its own viewId and is capable of navigating independently of other ViewPorts. A browser window and a region are both examples of a ViewPort. The RootViewPort displays the main page in a browser window. The RootViewPort may have child ViewPorts, for example, regions on the page, but does not have a parent ViewPort.

16.1.11 What You May Need to Know about Securing ADF Regions

You can set security on the ADF bounded taskflow that is displayed in an ADF region as well as on the page definition for the page containing the region (see [Chapter 28, "Adding Security to a Fusion Web Application"](#) for more information). If an end user displays a page that contains an ADF region he is not authorized to view, the contents of the ADF region will not display. No method for logging into the ADF region will be provided.

16.2 Creating ADF Dynamic Regions

An ADF dynamic region is similar to an ADF region, except that its task flow ID in the task flow binding is determined dynamically at runtime. This enables the application to determine at runtime which ADF bounded task flow to execute within the dynamic region. For example, a page displays employee information. A dynamic region added to the page displays a form for updating employee address information when the user clicks a button. Then the same region switches to a different form for updating the social security number when the employee clicks another button on the page.

Or, you could create a dynamic region that first displays a customer's street address. The end user might click a link or button to display forms for changing the billing address or the shipping address. The page fragment that displays the customer address and the two forms are all within the same ADF dynamic region, which saves space on the main page where the region is located.

As shown in [Example 16–4](#), the task flow binding for the ADF dynamic region contains an EL expression that returns the current `taskFlowId`. When the `taskFlow id` value changes, the dynamic region refreshes automatically. New input parameter values are automatically passed to it.

Example 16–4 Dynamic Task Flow Binding for ADF Dynamic Region

```
<taskFlow id="dynamicRegion1"
          taskFlowId="#{dynamicRegionBean.dynamicTaskFlowId}"
          xmlns="http://xmlns.oracle.com/adf/controller/binding"/>
```

When an ADF dynamic region is reinitialized, the application must reinitialize the task flow binding associated the region. This includes evaluating whether there are new input parameters and input parameter values to be passed to the ADF dynamic region.

When you drop an ADF bounded task flow onto a page to create an ADF dynamic region, you also specify any input parameters defined for the region in the task flow

binding in the page definition file. See [Section 16.1.3, "How to Specify Parameters for an ADF Region"](#)

16.2.1 How to Create an ADF Dynamic Region

The steps for adding an ADF bounded task flow to a page as an ADF dynamic region are similar to adding it as an ADF region.

In the following example, you might have an order page that displays products and services. If the user selects a product on a page, a details ADF dynamic region might show product details. If a service is selected, the same details region should show service details.

Before you begin:

- Create two page fragments that display product details and service details, for example, `ProductDetails.jsff` and `ServiceDetails.jsff` (see [Chapter 20, "Creating a Basic Databound Page"](#)).
- Create two ADF bounded task flows that use page fragments (see [Section 16.1.1, "How to Create an ADF Region"](#)), each containing one of the page fragments.
- Specify parameters (see [Section 16.1.3, "How to Specify Parameters for an ADF Region"](#)) for both bounded task flows.

To create a dynamic region:

1. Open a JSF page in the editor.
2. Drop the first ADF bounded task flow onto the JSF page as an ADF region. For more information, see [Section 16.1.1, "How to Create an ADF Region"](#).
3. In the Application Navigator, drag the second ADF bounded task flow onto the page and drop it anywhere on the JSF page.

A menu with choices for adding the ADF bounded task flow as a region or dynamic region displays. If your ADF bounded task flow is not eligible to be dropped as an ADF region (for example, it contains pages, not page fragments), you will not see this menu.

4. Click **Create > Dynamic Region**.

A dialog displays asking you to specify a managed bean. The managed bean will be used to store the value of the ID for the bounded task flow that is displayed within the dynamic region.

5. Select a managed bean in the **Managed Bean** drop-down list.
6. If no managed bean exists for the page, click the + icon next to the **Managed Bean** drop-down list to create a new one.
 - a. In the **Name** field, enter a name for the managed bean, for example, `dynamicRegionBean`.
 - b. In the **Class** field, enter the class for the managed bean.
 - c. In the **Scope** drop-down list, select a scope for the managed bean.

The managed bean is used to swap the different task flow definitions into the task flow binding of the ADF dynamic region.

For example, you could name the bean `DynamicTFBean` and specify `DynamicTF` as the class. This will generate `DynamicTF.java` with a property named `taskFlowId` which contains the id of the task flow that will be rendered in the

ADF dynamic region. Manipulating the value of this property (`taskFlowId`) at runtime enables you to switch between the ADF task flows in the dynamic region.

7. Click **OK**.
8. Add a method, for example, `setTaskFlowId(String param)` in `DynamicTF.java` to switch the value of `taskFlowId`. This method sets the value of the `taskFlowId` property to an ADF task flow parameter, for example `parm`.
9. The Edit Task Flow Binding dialog displays the input parameter definition for the bounded task flow you dropped on the page. See [Section 16.1.3, "How to Specify Parameters for an ADF Region"](#) for more information. Select the input parameter definition, for example, `parm`, and click **OK**.

When you are finished, a user should be able to click an order line in the page to invoke the method `setTaskFlowId`. This passes the corresponding `taskFlowId` as a input parameter value, which is used to specify the id of the ADF bounded task flow that should display.

16.2.2 How to Override Parameter Values

The various ADF bounded task flows that display within the ADF dynamic region can each specify different input parameter definitions. You can choose to list all of the input parameters defined for the bounded task flows within the `<parameters>` element in the task flow binding, as described in [Section 16.1.3, "How to Specify Parameters for an ADF Region"](#). In addition, you can optionally use the `<parameterMap>` element to identify which input parameters are used and which are overridden.

You specify the `<parameterMap>` element in the page definition. The position of the `<parameterMap>` element within the `<parameters>` list determines whether or not values in the `parameterMap` are used. The `<parameterMap>` element specifies an EL expression that returns a `java.Util.map` object. The object contains task flow input parameter name-value pairs. The parameter set for the task flow is built up from the order the values that are specified in the task flow binding.

The `<parameterMap>` element specifies an EL expression that returns a `java.Util.map` object. The object contains task flow input parameter name-value pairs. The parameter set for the task flow is built up from the order the values that are specified in the task flow binding.

Note: If you specify `Refresh="ifNeeded"`, parameters are not supported in the `<parameterMap>`. The only condition that determines if the region need to be refreshed is the boolean value returned by the evaluation of `RefreshCondition` (see [Section 16.1.7, "What You May Need to Know About Refreshing an ADF Region"](#) for more information).

In [Example 16–5](#), a parameter named `regionParm1` in the parameter map overrides the first parameter value specified. The parameters named `regionParm2` and `regionParm3` in the parameter map are overridden by the last two parameter values specified in the example.

Example 16–5 task flow binding for ADF Dynamic Region

```
<taskFlow id="Region1"
          taskFlowId="${RegionBean.TaskFlowId}"
```

```

    xmlns="http://xmlns.oracle.com/adf/controller/binding" />
<parameters>
    <parameter id="regionParm1" value="#{pageFlowScope.data.x}">
        <parameterMap="#{pageFlowScope.data.taskFlowParams}">
            <parameter id="regionParm2" value="#{pageFlowScope.data.y}">
                <parameter id="regionParm3" value="#{pageFlowScope.data.z}">
            </parameters>
        </taskflow>
    </parameters>
</taskflow>

```

16.3 Creating Dynamic Region Links

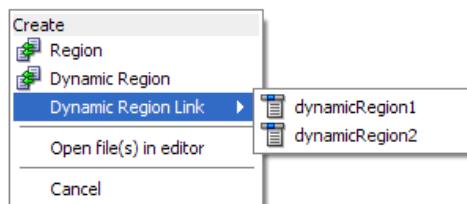
An ADF dynamic region link swaps an ADF bounded task flow for another ADF bounded task flow within an ADF dynamic region. An end user clicks a UI component such as a button or link to update the dynamic region with the new ADF bounded task flow. For example, the original ADF bounded task flow in the ADF dynamic region might display general information about an employee, for example an ID and photo. Clicking an ADF dynamic region link labeled **Details** updates the ADF dynamic region with a table containing more information about the employee.

When an end user clicks the UI component, a method is invoked on the dynamic region's managed bean. The value of the new ADF bounded task flow is passed to the method and the dynamic region is refreshed with the new ADF task flow. The referenced bounded task flow then displays within the dynamic region.

By default, a dynamic region links swaps another ADF bounded task flow in for the original, but cannot swap back to the original. To toggle back to the original ADF bounded task flow, you could add a second ADF dynamic region link on the page that, when clicked, swaps the current task flow back to the original one.

You can add a dynamic region link if you already have at least one ADF dynamic region on a page and are adding a new ADF bounded task flow as a dynamic region to the same page. After you drop the ADF bounded task flow on the page and choose to create a dynamic region link, a menu displays all of the dynamic regions currently on the page.

Figure 16–8 Dynamic Region Link Menu



You select the dynamic region within which you want to display the contents of the ADF bounded task flow. JDeveloper then adds a command link to the page, as shown in [Example 16–6](#). JDeveloper also updates the Java class for the dynamic region managed bean and updates the corresponding task flow binding with any new parameters.

Example 16–6 Dynamic Region Link

```
<af:commandLink text="region2" action="#{RegionBean.region2}"
    id="dynamicRegionLink1"/>
```

Tip: You can use the values in the dynamic region link in other UI components. For example, you could create a Selection List in which each of the items in the list links to a different ADF bounded task flow. All of the linked ADF bounded task flows would display in the same dynamic region. The links perform a method in the class behind the managed bean created to swap the ADF bounded task flow it displays.

The following steps assume you have completed the steps for creating an ADF dynamic region and adding it to a page, described in [Section 16.2.1](#) above.

To create a dynamic region link:

1. In the editor, open the JSF page you created in [Section 16.2.1](#) above.
2. In the Application Navigator, drag an ADF bounded task flow and drop it anywhere on the page.

The ADF bounded task flow must contain page fragments, not pages.

3. A menu with choices for adding the task flow as a dynamic region link displays.
4. Select **Dynamic Region Link**.

A menu display a list of all ADF dynamic regions that have already been added to the page.

5. Select the name of the dynamic region in which you want to display the contents of the ADF bounded task flow.
6. Click **OK**.

Creating Complex Task Flows

This chapter describes using advanced features of ADF task flows and includes the following sections:

- [Section 17.1, "Using Initializers and Finalizers"](#)
- [Section 17.2, "Managing Transactions"](#)
- [Section 17.3, "Reentering an ADF Bounded Task Flow"](#)
- [Section 17.4, "Handling Exceptions"](#)
- [Section 17.5, "Saving for Later"](#)
- [Section 17.6, "Creating a Train"](#)
- [Section 17.7, "Running an ADF Bounded Task Flow as a Modal Dialog"](#)
- [Section 17.8, "Creating an ADF Task Flow Template"](#)
- [Section 17.9, "Creating a Page Hierarchy"](#)
- [Section 17.10, "How to Specify Bootstrap Configuration Files"](#)
- [Section 17.11, "Using the adf-config.xml File to Configure ADF Controller"](#)

17.1 Using Initializers and Finalizers

An initializer is custom code that is invoked when an ADF bounded task flow is entered. You specify both the initializer and finalizer as an EL expression for a method on a managed bean, for example,

```
# {pageFlowScope.modelBean.releaseResources}.
```

There are two techniques for running initializer code at the beginning of the ADF bounded task flow, depending on whether the task flow may be reentered via a browser Back button:

- No reentry via back button expected: Designate a method call activity as the default activity (first to execute) in the ADF bounded task flow. The method call activity calls a custom method containing the intializer code. See [Section 14.5, "Using Method Call Activities"](#) and [Section 20.4.4, "What You May Need to Know About the Browser Back Button and Navigating Through Records"](#)for more information.
- Back button reentry possible: Specify an intializer method using an option on the ADF bounded task flow metadata (see [Section 13.2, "Creating Task Flows"](#)for more information). Use this technique if you expect that a user may reenter the task flow using a browser back button. In such a case, the designated default activity for the

bounded task flow may never be called, but a method designated using the Initializer method will.

A finalizer is custom code that is invoked when an ADF bounded task flow is exited via a task flow return activity or because an exception occurred. The finalizer is a method on a managed bean. Common finalizer tasks include releasing all resources acquired by the ADF bounded task flow and perform cleanup before exiting the task flow.

17.2 Managing Transactions

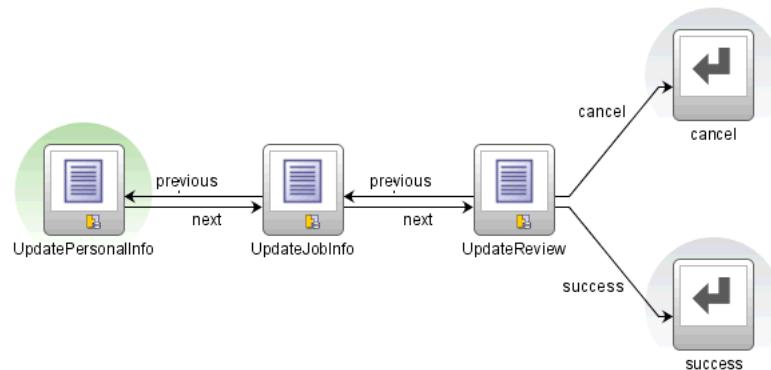
A transaction is a persisted collection of work that can be committed or rolled back together as a group. You can use an ADF bounded task flow to represent a transactional unit of work and to declaratively manage transaction boundaries. In the Fusion Order Demonstration application, the customer registration and employee registration task flows were both implemented with the use of task flow return activities. The **Cancel** button implements rollback in the task flows. The **OK** button on the review.jsff page fragment implements Commit functionality.

An end user can navigate from the Shopping Cart to initiate a backorder request for an out-of-stock item. The back order request application is implemented as an ADF bounded task flow that initiates a new transaction upon entry.

Transaction options on the called task flow definition specify whether a called ADF bounded task flow should join an existing transaction, create a new one, or create a new one only if there is no existing transaction.

If the called ADF bounded task flow is able to start a new transaction (based on the **transaction** option that you selected), you can specify whether the transaction will be committed or rolled back when the task flow returns to its caller. The commit and rollback options are set on the task flow return activity that returns control back to the calling task flow. The same task flow that starts a transaction must also resolve the transaction.

In a called task flow definition, you can specify two different return task flow activities that result in either committing or rolling back a transaction in the called ADF bounded task flow. Each of the task flow return activities passes control back to the same calling task flow. The difference is that one task flow return activity specifies the commit option, while the other specifies the rollback option. As shown in [Figure 17-1](#), if transaction processing successfully completes, control flow passes to the success task flow return activity, which specifies options to commit the transaction. If the transaction is cancelled before completion, the cancel task flow activity specifies options to roll back the transaction.

Figure 17-1 Task Flow Return Activities in Called ADF Bounded Task Flow

If no transaction option is specified, a transaction is not started on entry of the called ADF bounded task flow. A runtime exception is thrown if the ADF bounded task flow attempts to access transactional services.

If you want changes the user made within the called ADF bounded task flow to be discarded when it is exited, you can specify the **restore-save-point** option on the task flow return activity. The ADF Controller will roll back to the previous ADF Model save point that was created when the ADF bounded task flow was entered. The **restore-save-point** option applies only to cases when an ADF bounded task flow is entered by joining an existing transaction (either the **requires-existing-transaction** or **requires-transaction** option is also specified) and a Save Point is created upon entry.

Best Practice:

You can use a return activity to call the `commit` action or use a button bound to the `commit` action. If possible, use a task flow return activity. Using a task flow return activity will commit all data controls used by the ADF task flow. It also makes it easier to see where your application is doing commits and rollbacks, and is therefore easier to maintain.

You must use `invokeAction` if:

- You want to commit before the end of an ADF task flow.
 - You are not using the ADF controller.
 - The task flow uses multiple data controls and you do not want to commit all of them.
-

17.2.1 How to Enable Transactions in an ADF Bounded Task Flow

Before you begin, you should have two task flows. One calls the second ADF bounded task flow using a task flow call activity. The called ADF bounded task flow will specify a transaction option that defines whether or not a new transaction will be created when it is called. It will also contain a task flow return activity that specifies an outcome upon return to the caller.

To enable an ADF bounded task flow to run as a transaction:

1. In the editor, open the called ADF task flow.

2. In the Overview tab for the called ADF bounded task flow, click **Behavior** and expand the **Transaction** section.
3. Choose one of the following from the **transaction** drop-down list:
 - **requires-existing-transaction**: When called, the ADF bounded task flow participates in an existing transaction already in progress.
 - **requires-transaction**: When called, the ADF bounded task flow either participates in an existing transaction if one exists, or starts a new transaction upon entry of the ADF bounded task flow if one doesn't exist.
 - **new-transaction**: A new transaction is always started when the ADF bounded task flow is entered, regardless of whether or not a transaction is in progress. The new transaction is completed when the ADF bounded task flow exits.

If you do not choose any item, the called ADF bounded task flow does not participate in any transaction management.

Note: After choosing a transaction option, you may also need to check the **data-control-scope** option for the bounded task flow to determine if there are any interactions between the options. See [Section 15.3.1, "What You May Need to Know About Managing Transactions"](#) for more information.

4. If you choose **requires-existing-transaction** or **requires-transaction**, you can optionally select **true** in the **no-save-point** drop-down list.
If you select **true**, an ADF Model save point will not be created on task flow entry. An ADF Model save point is a saved snapshot of the ADF Model state. Selecting **true** means that overhead associated with a save point is not created for the transaction.
5. In the editor, select the task flow return activity in the called ADF bounded task flow.
6. In the Property Inspector, click **Behavior**.
7. If the called task flow definition supports creation of a new transaction (task flow definition specifies **requires-transaction** or **new-transaction** options), select one of the following in the **End Transaction** dropdown list:
 - **commit**: commits the existing transaction to the database.
 - **rollback**: rolls back a new transaction to its initial state on task flow entry. This has the same effect as cancelling the transaction.
8. In the **restore-save-point** drop down list, select **true** if you want changes the user made within the called ADF bounded task flow to be discarded when it is exited. The savepoint that was created upon task flow entry will be restored.

17.2.2 What Happens When You Specify Transaction Options

[Example 17-1](#) shows the metadata for transaction options on a called task flow definition. The `<new-transaction>` element indicates that a new transaction is always started when the ADF bounded task flow is entered.

Example 17-1 Called Task Flow Definition Metadata

```
<task-flow-definition id="trans-taskflow-definition">
  <default-activity>taskFlowReturn1</default-activity>
```

```

<transaction>
  <new-transaction/>
</transaction>
<task-flow-return id="taskFlowReturn1">
  <outcome>
    <name>success</name>
    <commit/>
  </outcome>
</task-flow-return>
</task-flow-definition>

```

[Example 17-1](#) also shows the metadata for transaction options on the task flow return activity on the called task flow. The `<commit/>` element commits the existing transaction to the database. The `<outcome>` element specifies a literal outcome, for example, `success`, that is returned to the caller when the ADF bounded task flow exits. The calling ADF task flow can define control flow rules based on this outcome to handle control flow upon return (see [Section 14.7, "Using Task Flow Return Activities"](#) for more information).

17.3 Reentering an ADF Bounded Task Flow

To deal with cases in which the end user clicks the back button to navigate back into an ADF bounded task flow that was already exited, you can specify **task-flow-reentry** options for the task flow definition. These options specify whether a page in the ADF bounded task flow can be reentered.

- **reentry-allowed:** Reentry is allowed on any view activity within the ADF bounded task flow.
- **reentry-not-allowed:** Reentry of the ADF bounded task flow is not allowed. If you specify reentry-not-allowed on a task flow definition, an end user can still click the browser back button and return to a page within the bounded task flow. However, if the user does anything on the page such as clicking a button, an exception (for example, `InvalidTaskFlowReentry`) is thrown indicating the bounded task flow was reentered improperly. The actual reentry condition is identified upon the submit of the reentered page.

You can set up an exception handler to display the exception and route control flow in order to navigate to the default activity of the called task flow definition. If the task flow definition was not called from another task flow definition, a normal web error is posted and handled as specified in the `web.xml` file.

- **reentry-outcome-dependent:** Reentry of an ADF bounded task flow using the browser back button is dependent on the outcome that was received when the same ADF bounded task flow was previously exited via task flow return activities. If specified, any task flow return activities on the called ADF bounded task flow must also specify either **reentry-allowed** or **reentry-not-allowed** to define outcome-dependent reentry behavior.

If you choose this option, the user can navigate to a task flow using a back button based solely on how the user originally exited the task flow. For example, a task flow representing a shopping cart can be reentered if the user exited by canceling an order, but not if the user exited by completing the order.

Upon reentry, ADF bounded task flow input parameters are evaluated using the current state of the application, not the application state existing at the time of the original ADF bounded task flow entry.

Note: Different browsers handle the back button differently. In order to ensure that back button navigation is properly detected across all browsers, the view activities within the task flow need to be properly configured. When a task flows uses the **reentry-not-allowed** or **reentry-outcome-dependent** option, the **redirect** attribute on each view activity within the task flow should be set to true. See [Section 14.3, "Using URL View Activities"](#) for more information on how to configure view activities.

17.3.1 How to Set Reentry Behavior

To set reentry behavior:

1. Open the ADF bounded task flow diagram in the editor.
2. Click the **Overview** tab.
3. Click **Behavior**.
4. In the **Task Flow Reentry** list, choose one of the following:
 - **reentry-allowed**
 - **reentry-not-allowed**
 - **reentry-outcome-dependent**

17.3.2 How to Set Outcome-Dependent Options

The following steps apply only to ADF bounded task flows that have specified the **reentry-outcome-dependent** option.

1. In the ADF bounded task flow, add a task flow return activity. (See [Section 14.7, "Using Task Flow Return Activities"](#) for more information).
2. In the task flow diagram, select the task flow return activity.
3. In the Common page of the Property Inspector, expand the **Outcome** section.
4. In the **name** field, enter the name of literal outcome, for example, `success` or `failure`.
5. In the Property Inspector, click **Behavior**.
6. In the **Reentry** drop-down list, select either:
 - **reentry-allowed**
 - **reentry-not-allowed**

17.3.3 What You Should Know About Managed Bean Values Upon Task Flow Reentry

When an end user reenters an ADF bounded task flow using the browser Back button and reentry is allowed (either by setting **reentry-allowed** on the bounded task flow or a combination of **reentry-outcome-dependent** on the bounded task flow and **reentry-allowed** on the task flow return activity as described in [Section 17.3.1](#) and [Section 17.3.2](#)), the value of a managed bean on the reentered task flow is set back to its the original value, the same value that it was before the end user left the parent task flow.

This results in the managed bean value resetting back to its original value before a view activity in the reentered task flow renders. Any changes that occurred before reentry are lost. To change this behavior, specify the `<redirect>` element on the view activity in the reentered bounded task flow. When the end user reenters the bounded task flow using the Back button, the managed bean has the new value from the parent task flow, not the original value from the child task flow that is reentered.

17.4 Handling Exceptions

During execution of an ADF task flow, exceptions can occur that may require some kind of exception handling, for example:

- Method call activity throws an exception
- Custom method you have written as a task flow intializer or finalizer throws an exception
- User is not authorized to execute the activity

To handle exceptions thrown from an activity or caused by some other type of ADF Controller error, you can create an exception handler for an ADF bounded or unbounded task flow.

When an exception is raised within the task flow, control flow passes to the designated exception handling activity. For example, the exception handling activity might be a view that displays an error message. Or the activity might be a router that passes control flow to a method based on an EL expression that evaluates the type of exception, for example,

```
# {someException="oracle.adf.Controller.ControllerException"}.
```

After control flow passes to the error handling activity, flow from the activity uses standard control flow rules (see [Section 13.1.3, "Control Flows"](#) for more information).

You can optionally specify an exception handler for both ADF bounded and unbounded task flows. If specified, the task flow can have only a single exception handler. However, a task flow called from inside another task flow can have a different exception handler than the one for the caller. In addition, a region on a page can have a different exception handler than the task flow containing the page. The exception handler activity can be any supported activity type, for example, a view or router activity.

If an ADF bounded task flow does not have a designated exception handler activity, control passes to an exception handler activity in a calling ADF bounded task flow, if there is a calling task flow and it contains an exception handler activity. The exception is propagated up the task flow stack until either an exception handler activity or the top-level ADF unbounded task flow is reached. If no exception handler is found, the exception is propagated to the web container.

You can designate a exception handler activity for an ADF bounded task flow running as an ADF region. If an exception occurs in the bounded task flow and it is not handled by the task flow's exception handler, the exception is not propagated up the task flow stack of the parent page. Instead, it becomes an unhandled exception.

To designate an activity as an exception handler for a task flow:

1. Right-click the activity in the task flow diagram, then choose **Mark Activity > Exception Handler**.

A red exclamation point is superimposed on the activity in the task flow to indicate it is an exception handler.



2. To unmark the activity, right-click the activity in the task flow diagram, then choose **Unmark Activity > Exception Handler**.

If you mark an activity as an exception handler in a task flow that already has a designated exception handler, the old handler is unmarked.

17.4.1 Handling Exceptions During Transactions

As a best practice, any ADF bounded task flow representing a managed transaction should designate an exception handling activity. When running an ADF bounded task flow managed transaction, the ADF Controller attempts to commit the transaction when it reaches a task flow return activity identified in metadata for a commit (see [Section 17.2, "Managing Transactions"](#) for more information). If the commit throws an exception, control is passed to the ADF bounded task flow's exception handling activity.

This provides the end user with a chance to correct any data she may have added on a page and then reattempt to commit it, for example, by clicking a Save button. You can use the exception handling activity (for example, a view activity) to display a warning message on a page that tells the user to correct the data and attempt to commit it again.

17.4.2 What Happens When You Designate an Exception Handler

After you designate that an activity is the exception handling activity for a task flow, the task flow metadata updates with an `<exception-handler>` element that specifies the id of the activity, as shown in [Example 17–2](#).

Example 17–2 `<exception-handler>` element

```
<exception-handler>MyView</exception-handler>
```

17.4.3 What You May Need to Know About Handling Validation Errors

For validation errors on a JSF page, you can rely on standard JSF to attach validator error messages to either specific components on a page or the whole page. A component-level validator typically attaches an error message inline with the specific UI component. There is no need to pass control to an exception handler activity.

In addition, Oracle recommends that your application define validation logic on data controls that are executed during the Validate Model Updates phase of the JSF lifecycle. By doing so, data errors are found as they are submitted to the server without waiting until attempting the final commit.

Validations done during the Validate Model Updates typically do not have direct access to the UI components because the intention is to validate the model after the model has been updated. These validations are often things like checking if dependent fields are in sync. In these cases, the error message is usually attached to the whole page, which this logic can access.

Errors detected during the Validate Model Updates phase should be attached to the JSF page, and `FacesContext.renderResponse()` should be called. This signals that following this phase, the current (submitting) page should be rendered showing the attached error messages. There is no need to pass control to an exception handler activity.

For more information, see [Chapter 8, "Implementing Validation and Business Rules Programmatically"](#).

17.5 Saving for Later

Saving for later captures a snapshot of a Fusion web application at a specific instance called a save point. This allows you to retain the current state of the task if a user leaves a page without finalizing it. For example, an end user may need to stop in the middle of completing a page containing an employee profile form because it is the end of the business day or because additional information is required to answer questions.

You can use the save point restore activity to restore whatever was captured at the original save point. This enables the end user to complete the rest of the task at a future date, with former unsaved values restored to the page. For example, you could create a page containing a drop-down list of save point ids and a button. When an end user selects a save-point-id and clicks the button, the application executes the save point restore activity. See [Section 14.8, "Using Save Point Restore Activities"](#) for more information.

There are two categories of saving for later:

- **Explicit**

For example, a page contains a button that the user can click to save all data entered so far on the page. See [Section 17.5.2, "How to Restore Save Points"](#) for more information.

Explicit save for later is available for both ADF unbounded and bounded task flows.

- **Implicit**

For example, a user accidentally closes a browser window without saving data entered on a page, a user logs out without saving the data, or the session times out. See [Section 17.5.4, "How to Enable Implicit Save For Later"](#) for more information.

Implicit save for later can only originate from a bounded task flow. To enable implicit savepoints, the `enable-implicit-savepoints` property must be set to true in the `adf-config.xml` configuration file. The example below shows how to set up `adf-config.xml` to enable implicit save point capability within an application. See [Section 17.11, "Using the adf-config.xml File to Configure ADF Controller"](#) for more information.

```
<adf-config xmlns="http://xmlns.oracle.com/adf/config">
    <adfc-controller-config xmlns=
        "http://xmlns.oracle.com/adf/controller/config">
        <enable-implicit-savepoints>true
        </enable-implicit-savepoints>
    </adfc-controller-config>
</adf-config>
```

Implicit save points are generated only if a critical task flow is present in any of the page flow stacks for any view port under the current root view port. An implicit

save point is not generated if the request is a request for an ADF Controller resource, such as:

- task flow call activity
- task flow return activity
- save point restore activity
- a dialog

Implicit save points are deleted when the task flow at the bottom of the stack completes or a new implicit save point is generated, whichever comes earlier.

You can call the `createSavePoint()` API to create a save point, identified by its `save-point-id`. A save point captures a snapshot of an application at a specific instance. For an explicit save, this includes everything from the time the save point is created in the originating task flow down through the call stack. Explicit save points are automatically removed when an ADF bounded task flow at the bottom of the task flow call stack completes. `createSavePoint()` is located under ADF Controller Objects -> controllerContext -> savePointManager.

For an implicit save, this includes everything from the time the save point is created in the originating task flow, both down and up the call stack.

Implicit save points are created for an application only if you have set the `enable-implicit-savepoints` property to `true` in the `adf-config.xml` configuration file. The save point and application state information are saved in a database.

State information for the application at the save point is also saved, as shown in [Table 17-1](#).

Table 17-1 Saved Application State Information

Saved State Information	Description
User Interface State	UI state of the current page, including selected tabs, selected checkboxes, selected table rows, and table column sort order. This assumes the end user cannot select the browser back button on save point restore.
Managed Beans	State information saved in several possible memory scopes, including session and pageFlow scope. The managed beans must be serializable in order to be saved. If you have <code>pageFlowScope</code> beans that are not serializable and you attempt to create a save point, a runtime exception occurs. Request scope is not supported since its lifespan is a single HTTP request and can't be used to store cross request application state. Save for later will not save and restore application scoped managed beans since they're not passivated in failover scenarios. Therefore, the application is always responsible for ensuring that all required application-scoped state is available. Potential naming conflicts for managed beans already existing within the session scope at restore time will not occur because multiple managed beans using the same name shouldn't be implemented within an application.

Table 17-1 (Cont.) Saved Application State Information

Saved State Information	Description
Navigation State	Task flow call stack, which ADF Controller maintains as one task flow calls another at runtime.
ADFm Model State	Fusion web applications use ADFm to represent the persisted data model and business logic service providers. The ADFm model holds any data model updates made since beginning the current bounded task flow. The model layer determines any limits on the saved state lifetime. See Chapter 36, "Application State Management" for more information.

The same save-point-id can be used when the same user repeatedly performs a save for later on the same instance of a task flow within the same browser window within the same session. If a user performs a save for later following every page of a task flow, only one save-point-id will be created. It is overlaid each time a save for later is performed.

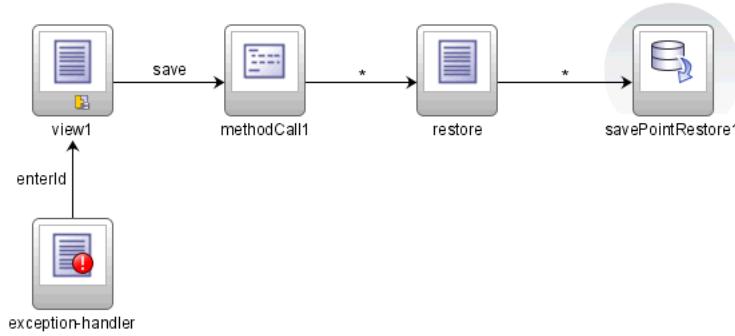
17.5.1 How to Add Save For Later Capabilities

To enable save for later, you must call the `createSavePoint()` method to create a save-point-id. Later, you use the save point restore activity to restore application state and data associated with a save-point-id (for more information, see [Section 17.5.2, "How to Restore Save Points"](#)).

A save point is stored in a database table. The table used to store save points is called ORADFCSAVPT. If this table does not exist within the database, it will automatically be created along with the first save point. When the save point is restored, the save point is deleted from the database.

[Figure 17-2](#) contains an example that illustrates how you can restore a previously created save point in a task flow:

- view1 contains a button that the end user can click to perform a Save for Later
- The method call activity calls `createSavePoint` (see [Example 17-3](#)). The `createSavePoint` method adds a save-point-id. The save-point-id captures everything up to the point in the taskflow where `createSavePoint` is called.
- The restore view activity contains an input text field into which a user can enter a save-point-id and click a button to pass the save-point-id to the save point restore activity.
- `savePointRestore1` is a save point restore activity that uses the `save-point-id` to restore the application state and data that was captured when the save point was originally created.

Figure 17–2 Save For Later Task Flow

You can access the `createSavePoint` method under the `currentViewPort` node under ADF Controller Objects. Instead of calling this method directly, however, you may want to want to create your own method that calls `createSavePoint`, as shown in [Example 17–3](#). If you do this, you can update the save-point-id with any custom attributes you create on your method.

Example 17–3 Example Custom Method For Creating save-point-id

```

package viewController;

import java.io.Serializable;
import oracle.adf.Controller.ControllerContext;
import oracle.adf.Controller.ViewPortContext;

public class SaveForLater implements Serializable
{
    public SaveForLater()
    }

    public String saveTaskFlow()
    {
        ControllerContext cc = ControllerContext.getInstance();
        if (cc != null)
        {
            SavePointManager mgr = cc.getSavePointManager();
            if (mgr != null)
            { String id = mgr.createSavePoint();
                System.out.println("Save point is being set "+id);
            }
        }
    }
  
```

The following steps describe adding a call to a custom method to a task flow similar to [Figure 17–2](#). The method creates a save point id similar to [Example 17–3](#).

To add save for later capability to a task flow:

1. Drag a method call activity from the ADF Task Flow drop-down list of the Component Palette and drop it on the diagram for the ADF bounded task flow.

2. In the diagram, select the Method Call activity.
3. In the Common page of the Property Inspector, enter an **id** for the method, for example, `callCreateSavePoint`.
4. In the **method** field, enter an EL expression for the save point method, for example,
`#{ControllerContext.getInstance().getSavePointManager.createSavePoint}`.
If you created your own method to call `createSavePoint`, enter that method name instead.
5. After adding the method call activity to the diagram, connect it to the other existing activities in the diagram using control flows (see [Section 13.2.3, "How to Add Control Flows"](#) for more information).
6. You can specify settings in the `adf-config.xml` file such as the save point default expiration time and where save points are stored (see [Section 17.11.1, "Specifying Save for Later Settings"](#) for more information).

17.5.2 How to Restore Save Points

Use the save point restore activity to restore a previously persisted save point for an application. To identify which save point should be restored, the save point restore activity uses the savepoint that was originally created using the `createSavePoint` method.

You can obtain a list of the current persisted savepoints using `createSavePoint`. However, ADF Controller does determine which savepoints to restore. A user must select the savepoint from a list or the application developer select it programmatically. The savepoint id is then passed to a save point restore activity to perform the restore.

To add a save point restore activity to an ADF bounded or unbounded task flow:

1. In the editor open the bounded or unbounded task flow where you want to add the save point restore activity.
2. Drag a save point restore activity from the ADF Task Flow drop-down list of the Component Palette and drop it on the diagram for the task flow.
3. In the diagram, select the save point restore activity.
4. In the Common page of the Property Inspector, enter an EL expression in the **save-point-id** field that, when evaluated, specifies the save-point-id that was created when the `createSavePoint` method was originally called. For example, you could enter `#{pageFlowScope.savepoint1}`.

17.5.3 How to Use the Save Point Restore Finalizer

When using the save point restore activity, you may need to include application-specific logic that will be performed as part of restoring application state.

Therefore, each task flow definition can specific a finalizer method via a method binding EL expression. The method is invoked after the task flow definition's state has been restored. It performs any necessary logic to ensure the application's state is correct before proceeding with the restore. For example, the application developer must provide restoration logic for validations of a temporal nature.

17.5.4 How to Enable Implicit Save For Later

An implicit save can occur when data is saved automatically because:

- the session times out due to end user inactivity
- the user logs out without saving the data
- the user closes the only browser window, thus logging out of the application

Navigation away from the current application using control flow rules (for example, using a goLink to an external URL) and having unsaved data is a valid condition for creating an implicit breakpoint.

Note: Implicit save points are created for an application only if you have set enable-implicit-savepoints to true in the `adf-config.xml` configuration file (see [Section 17.11, "Using the adf-config.xml File to Configure ADF Controller"](#) for more information). By default, `enable-implicit-savepoints` is set to false.

You can specify implicit save for later in a bounded task flow by selecting the **critical** checkbox for the task flow definition. By default, **critical** is not selected.

If multiple windows are open when the implicit save point is created, a different save-point-id is created for each browser window. This includes everything from the root view port of the browser window and down. You can use `ControllerContext.savePointManager.listSavePointIDs` to return a list of implicitly created save points.

To enable an implicit save:

1. In the editor for the ADF bounded task flow, click the **Overview** tab.
2. Click **Behavior** and then select the **critical** checkbox.

17.5.5 How to Set the Time-to-Live Period

The time-to-live period is the time between when a save point is first created and when it is removed by the Save Point manager. The default is 24 hours. You can specify the default time-to-live and other Save For Later configuration options in the `adf-config.xml` configuration file. If you specify DATABASE, you must call an API to remove save points. See [Section 17.11.1, "Specifying Save for Later Settings"](#) for more information.

You can change the time-to-live period by a call to the `setSavePointTimeToLive` method. The method syntax is shown in [Example 17–4](#).

Example 17–4 SetSavePointTimeToLive Method Syntax

```
void setSavePointTimeToLive(long timeInSeconds)
```

`timeInSeconds` is set as equal to or less than zero, the default time-to-live is used.

Tip: You typically will make this method call at the same time a save-point-id is initially created. To do this, you can include the call to `setSavePointTimeToLive` in the same custom method that you can create to call `createSavePoint` (see [Example 17–3](#)).

17.5.6 What You Need to Know about Using Save for Later with BC4J

To use the Save for Later feature with a BC4J application, you must change the `jbo.locking.mode` from pessimistic to optimistic. Pessimistic locking, the default setting, causes an old session to lock until the session has timed out. For example, you may run an application, change data without committing changes to the database. When you create a save point and try to restore it, you may get an error.

To set optimistic locking mode:

1. In the Application Navigator, right-click the BC4J Application Module and choose **Configurations**.
2. Click **Edit**.
3. Click the **Properties** tab.
4. Change `jbo.locking.mode` from **pessimistic** to **optimistic**.

17.6 Creating a Train

A train represents a progression of related activities that guides an end user to the completion of a task. The end user clicks a series of train stops, each stop linking to a particular page. [Figure 17–3](#) shows a train in the Fusion Order Demonstration application that is called when an end user clicks the **Self-register** link on the User Registration page.

Figure 17–3 Self-registration Train in Fusion Order Demonstration Application

This train contains four stops, each corresponding to a JSF page where the end user can enter and review registration information. The train stops are:

- Basic Information
- Address
- Payment Options
- Review

Each JSF page in the train contains an ADF Faces **Train** UI component that enables the user to navigate through the train stops in an order specified in the underlying train model.

Figure 17–4 Train UI Component



The optional **Train Button Bar** component ([Figure 17–5](#)) contains buttons that provide an additional means to navigate backwards and forwards through the stops. This component can be used in conjunction with the train component to provide multiple ways to navigate through train stops.

Figure 17–5 Train Button Bar UI Component



17.6.1 ADF Bounded Task Flows as Trains

You can create a train based on activities in an ADF bounded task flow that specifies the `<train/>` element in its metadata. You cannot create a train from activities in an ADF unbounded task flow.

Tip: You can also create an ADF task flow template that has the `<train/>` element in its metadata, then create an ADF bounded task flow based on the template.

Each bounded task flow can have a single train only. If the bounded task flow logically includes multiple trains, you must add each train to a separate ADF bounded task flow.

[Figure 17–6](#) displays the bounded task flow that the Fusion Order Demonstration application uses to create the Self-registration train shown in [Figure 17–3](#).

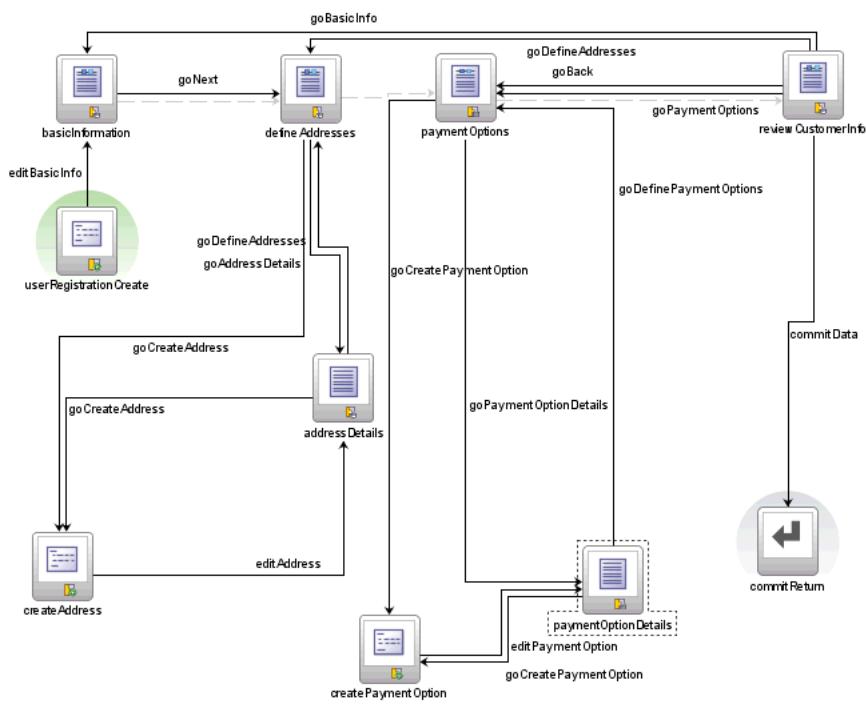
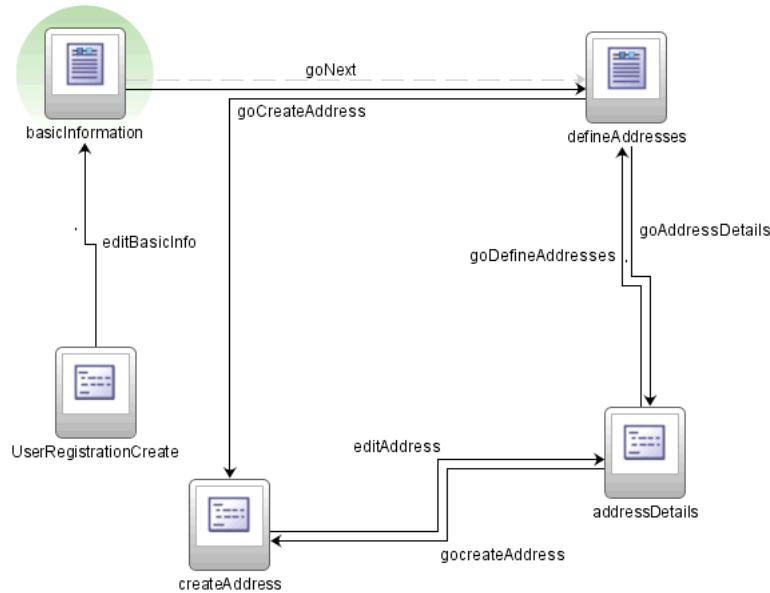
Figure 17–6 customer-registration-task-flow

Figure 17–7 contains a simplified detail of the first two train stops in customer-registration-task-flow. In the figure, an icon (two blue circles) identifies each train stop in the train. The dotted line connecting each stop indicates the sequence in which the end user visits them (see [Section 17.6.2, "Train Sequences"](#) for more information). Although you can't drag the dotted line to change the sequence, you can right-click on a train stop and choose options to move the train stop forwards or backwards in the sequence. Each train stop is usually a view activity corresponding to a JSF page although you can also add use a task flow call activity as a train stop.

Figure 17–7 Detail of customer-registration-task-flow

The task flow call activity is used to group sets of activities together as a train stop or to call a child train. For example, there are cases when other activities, such as router and method call activities, should be considered part of a train stop along with the corresponding view activity. A method call activity might need to precede a view activity for initialization purposes. When grouped this way, the activities can be performed as a set each time the train stop is visited. See Section [Section 17.6.4, "What You May Need to Know About Grouping Activities"](#) for more information.

Branching using router activities and control flow cases is supported on the task flow diagram containing the train as well as in child bounded task flows called from the train. See [Section 17.6.7, "What You May Need to Know About Branching"](#) for more information.

17.6.2 Train Sequences

By default, train stops are sequential. A user can select a sequential train stop only after visiting the train stop that is before it in the sequence. A nonsequential train stop can be performed in any order.

As shown in [Figure 17–8](#), a single train can contain both sequential and nonsequential stops. In the figure, First Step is the current train stop. Second Step is sequential because the user can click it only after visiting First Step. Third Step is nonsequential because it can be clicked immediately after the user visits First Step without also having to visit Second Step.

When First Step is the current stop, the end user cannot click the Fifth and Sixth Steps 6, indicating that they are sequential.

Figure 17–8 Train with Sequential and NonSequential Stops

You can set the sequential option on the view activity (or task flow call activity) for each train stop to specify whether it has sequential or nonsequential behavior. The sequential option contains an EL expression that evaluates end user input or some other factor, for example, `# {myTrainModel.isSequential}`. When the EL Expression evaluates to true, the train stop behaves as sequential. When it evaluates to false, the train stop behaves as nonsequential.

In addition, you can alter the overall train sequence by skipping over individual train stops. The skip option on the activity corresponding to the train stop uses an EL expression that evaluates end user input or some other factor to determine whether to skip over the train stop. If it evaluates to true, the train stop appears disabled and will be passed over. The end user is taken to the next enabled stop in the train. In addition, the train stop will be skipped if the end user clicks a **Next** or **Previous** button.

Note: If you want the train to execute by default at some train stop other than the first one, use the skip option. For example, if you want the train to begin executing at train stop 3, specify skip on train stops 1 and 2. This is a better practice than marking a view activity associated with a train stop as the default activity for the ADF bounded task flow, which can cause unpredictable results. For more information, see [Section 13.4, "Designating a Default Activity in an ADF Bounded Task Flow"](#).

17.6.3 How to Create a Train

To use an ADF bounded task flow as a train, its metadata must contain the `</train>` element in its task flow definition. There are several ways to include this element in the metadata:

- Select the **Create Train** checkbox in the Create ADF Task Flow dialog box (see [Section 13.2, "Creating Task Flows"](#) for more information).
- Select the **Create Train** checkbox in the Create ADF Task Flow Template dialog box, then use the template to create a new ADF bounded task flow (see [Section 17.8, "Creating an ADF Task Flow Template"](#) for more information).t
- Right-click on an existing bounded task flow diagram in the editor and choose **Train > Create Train**.
- Open an existing ADF bounded task flow in the editor, click the **Overview** tab, click **Behavior**, and select the **Train** checkbox

To create a train:

1. In the editor, open an ADF bounded task flow that is usable as a train.
You must include the `<train>` element in the ADF bounded task flow's metadata by following one of the methods described above.
2. Drag each view activity or JSF page you want to include in the train from the Application Navigator to the ADF bounded task flow diagram.
 - If you drag a JSF page, JDeveloper automatically adds a view activity to the diagram.
 - If you drag a view activity to the ADF bounded task flow diagram, you can double-click it later to create a new JSF page.

Tip: Don't worry adding pages or view activities to the diagram in a particular order. You can adjust the train sequence later.

3. To rearrange the order of train stops, right-click its corresponding activity in the diagram. Choose **Train**, and then choose a menu item to move the train stop within the sequence, for example, **Move Forward**, **Move Backward**, etc.
 4. By default, trains stops are sequential. You can optionally define whether the train stop behaves nonsequentially.
 - a. In the bounded task flow diagram, select the activity associated with the nonsequential train stop.
 - b. In the Property Inspector for the activity, click **Train Stop**.
 - c. In the **sequential** field, enter an EL expression, for example,
#{myTrainModel.isSequential}.

You can also specify a literal value, for example, `true`.
If the EL expression evaluates to true, the train stop will be sequential. If false, the train stop will be nonsequential.
 5. By default, a train stop will not be skipped. You can optionally designate that the train stop can be skipped based on the result of an EL expression.
 - a. In the ADF bounded task flow diagram, select the activity associated with the nonsequential train stop.
 - b. In the Property Inspector, click **Train Stop**.
 - c. In the **skip** field, enter an EL expression, for example,
#{myTrainModel.shouldSkip}.

If the EL expression evaluates to true, the train stop will be skipped. If false, the train stop will not be skipped.
 6. In the diagram, double-click each view activity that is being used as a train stop.
Double-clicking the view activity opens a dialog to create a new JSF page. If a JSF page is already associated with the view activity, the existing page displays.
 7. Open the JSF page in the editor.
 8. For each JSF page in the train, select a **Train** and, optionally, a **Train Button Bar** UI component from the Common Components section of the ADF Faces page of the Component Palette. Drag the UI component onto the JSF page.
- The Train and Train Button Bar UI components are not automatically added to pages and page fragments corresponding to the view activities within a task flow definition for a train. You must add them manually to each page or page fragment. You can also add them using a page template.
- After you add the components to the page, they are automatically bound to the train model. For more information about creating a train model, see the *Oracle Fusion Middleware Web User Interface Developer's Guide for Oracle Application Development Framework* for more information.

17.6.4 What You May Need to Know About Grouping Activities

Activities such as routers and method calls can be grouped with a view activity to form a single train stop. In the customer registration bounded task flow (Figure 17–6 above), the `createAddress` method call activity and the `addressDetails` view activity are grouped with the `defineAddress` view activity to create the second stop in the train. The activities are performed as a set each time the train stop is visited. `createAddress` validates user input on the Address page and `defineAddress` is an optional page where the end user can enter additional address information.

Because the page is optional and is accessed using a link on the Address page, it is not included in the bounded task flow as a separate train stop. Instead, it is grouped as one of the activities for the `createAddress` train stop.

Note: The recommended approach for grouping a set of activities to form a train stop is to use a child ADF bounded task flow (see [Section 17.6.5, "What You May Need to Know About Grouping Activities in Child Task Flows"](#)). The advantage of this approach is that all activities in the group are always performed together regardless of whether the train stop is being visited for the first time or on later returns.

If you don't use a child ADF bounded task flow, all activities from the leading non-view activity through the next view activity will be considered part of the train stop's execution.

Although the recommended approach is to group a set of activities for a train stop within a child bounded task flow, you can provide the same functionality without a call to a child bounded task flow. Instead you can group the activities on the parent bounded task flow, as shown in [Figure 17-9](#) below. All activities leading from the first non-view activity through the next view activity are considered part of the implied train stop's execution.

To group activities without a child bounded task flow, you must ensure that:

- all non-view and non-task flow call activities for the train stop, such as routers and methods calls, should follow the view or task flow call activity being used as the train stop.

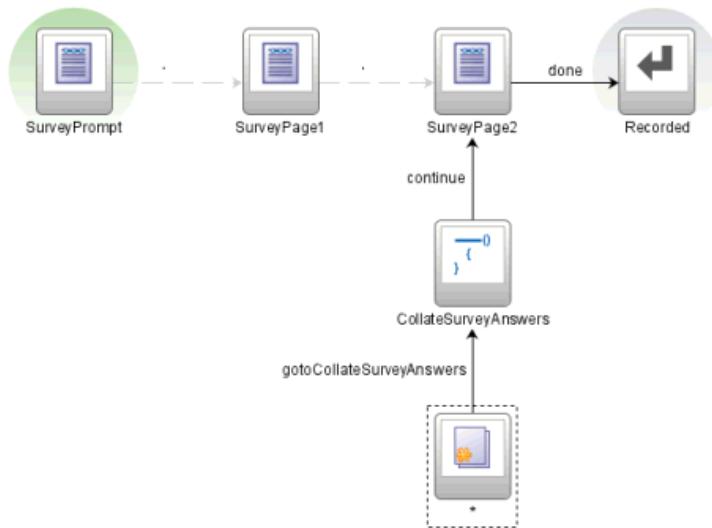
This associates the activities with the train stop and not with the previous stop in the train.

- a wildcard control flow leads to first activity of the train stop.

You must specify a value for **from-outcome** (for example `gotoCollateSurveyAnswers`) on the control flow leading from the wildcard control flow. This value must match the value you specify for the train **outcome** on the view activity that is used as the train stop.

The wildcard control flow ensures that if an end user returns to a previously visited trainstop, all activities will be performed beginning with the activity the wildcard control flow points to.

Figure 17–9 Grouping Activities Without Using a Task Flow Call Activity



In [Figure 17–9](#) above, the SurveyPage2 train stop consists of the following group of activities that execute in the following order:

1. Wildcard control flow leading to first activity of the train stop, CollateSurveyAnswers.

Note: The train stop **outcome** element is used only if an activity such as a method call activity or router is placed prior to the view activity train stop. The element allows you to specify a custom outcome that results in navigation to the train stop.

2. Method call to CollateSurveyAnswers method
3. SurveyPage2 view

17.6.5 What You May Need to Know About Grouping Activities in Child Task Flows

You can also group related activities along with a corresponding view activity in a child ADF bounded task flow. Then you can designate a task flow call activity as train stop in the train ([Section 14.6, "Using Task Flow Call Activities"](#) for more information). The task flow call activity calls the child ADF bounded task flow. The group of activities are always performed together regardless of whether the train stop is being visited for the first time or on later returns.

Best practice:

When grouping activities in a called child bounded task flow, non-view activities such as routers and methods calls typically precede the view activity in the control flow. You can include multiple view activities within the child bounded task flow, although in most cases there will be only one.

To group activities in the called child bounded task flow, you must ensure that:

- All non-view activities for the train stop, such as routers and methods calls, precede the view activity in the control flow of the child bounded task flow.
- the child task flow contains a single view activity, unless the view activities are dialogs or helper pages originating from the main train stop view activity.

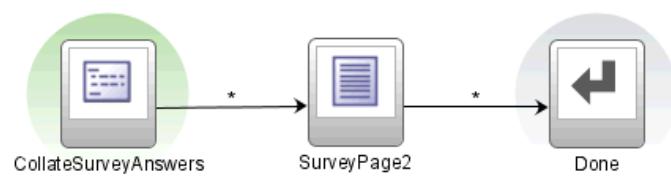
[Figure 17-10](#) shows the SurveyTaskFlow train stop, a task flow call activity that calls a child bounded task flow.

Figure 17-10 Task Flow Call Activity Train Stop



The child bounded task flow is shown in [Figure 17-19](#)

Figure 17-11 Called Child Bounded Task Flow



All of the activities in the child bounded task flow are performed together every time the SurveyTaskFlow train is visited, regardless if the end user is visiting the first time or later.

If you use a child ADF bounded task flow to group a set of activities, you must add a task flow return activity to the child task flow. The task flow return activity leads back to the parent bounded task flow, thus continuing the train. The task flow return activity should specify a value in **outcome**, for example, done, that will be used when returning to the parent train. In addition, you must manually add a control flow to the parent task flow that will be used to continue control flow within the train after returning from the child task flow.

As shown in [Figure 17-10](#) and [Figure 17-11](#) above, the value specified in the **from-outcome** for the control flow case (done) matches the task flow return activity **outcome** value.

17.6.6 What You May Need To Know About Using Child Trains

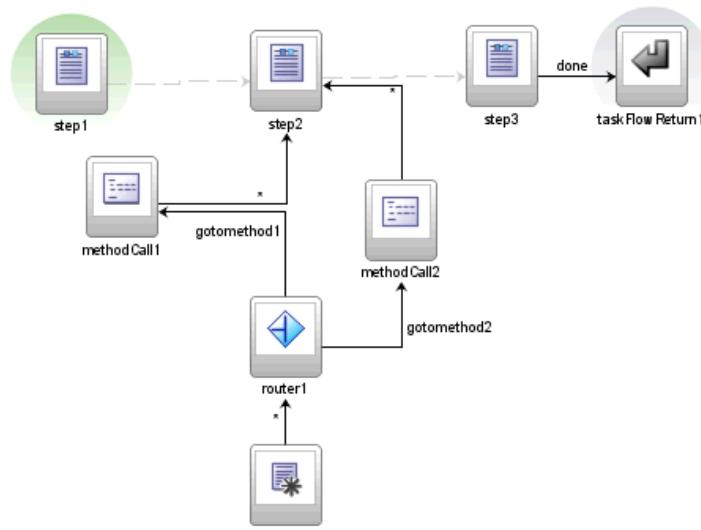
A train can use a task flow call activity as a train stop to invoke a child ADF bounded task flow representing another train. The child ADF bounded task flow must be created as its own train (must have the <train/> element in its metadata) and contain its own train stops. There is no limit to the depth of calls that are allowed to child trains.

17.6.7 What You May Need to Know About Branching

You can branch using router activities and control flow cases within a group of activities that are grouped to represent a single train stop, for example, the wildcard control flow rule router, and methods calls under step 2 in [Figure 17–12](#).

You cannot branch between the activities that represent each train stop in a train. For example, you can not branch between steps 1, 2, and 3 in [Figure 17–12](#).

Figure 17–12 Branching within Grouped Activities for a Single Train Stop



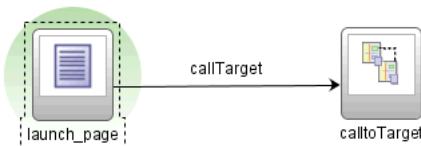
17.7 Running an ADF Bounded Task Flow as a Modal Dialog

An executing application can link to a page to get information from the user and then return to the original page to use that information. You can optionally display the page in a modal dialog that has to be closed before the end user can return to using the parent application.

Note: An ADF unbounded task flow must run in a secondary window, not a modal dialog.

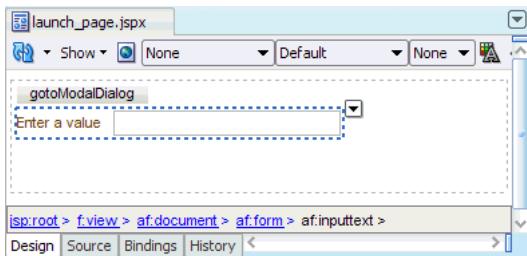
ADF bounded task flows can run in ADF regions. The ADF region can be in an af :popup UI component.

Figure 17–13 Source Task Flow



Suppose you created a task flow containing a view and task flow call activity similar to [Figure 17–13](#). When a user clicks a button in the `launch_page` view activity, the bounded task flow associated with `calltoTarget` executes.

Figure 17–14 `launch_page`



On `launch_page`, the outcome identified by a button's action property, `callTarget` passes control to the task flow call activity. If you select the `run-as-dialog` option on the task flow call activity, all of the pages within the called ADF bounded task flow execute within a modal dialog.

The value that the user enters in the `launch_page` can be used by the called task flow. For example, you could add a page on the called bounded task flow that displays the value in an output text component. The bounded task flow running as modal dialog can also pass values back to the caller.

Note: If your application uses ADF Controller features (for example, ADF task flows), specifying `dialog:` syntax in navigation rules within `faces-config.xml` is not supported.

However, you can use the `dialog:` syntax in the control flow rules that you specify in the `adfc-config.xml` file.

For example, you can specify the following in `adfc-config.xml`:

```
<?xml version="1.0" encoding="windows-1252" ?>
<adfc-config xmlns="http://xmlns.oracle.com/adf/Controller">
    <view id="view1">
        <page>/view1.jspx</page>
    </view>
    <view id="dialog">
        <page>/dialog/untitled1.jspx</page>
    </view>
    <control-flow-rule>
        <from-activity-id>test</from-activity-id>
        <control-flow-case>
            <from-outcome>dialog:test</from-outcome>
            <to-activity-id>dialog</to-activity-id>
        </control-flow-case>
    </control-flow-rule>
</adfc-config>
```

17.7.1 How to Run an ADF Bounded Task Flow in a Modal Dialog

Before you begin, create a source that contains the activities shown in [Figure 17–13](#). The view in the source task flow should be associated with a page containing an ADF Faces button and an input text field, similar to `launch_page` in [Figure 17–14](#).

To run an ADF bounded task flow as a modal dialog box:

1. In the Application Navigator, double-click the page that will launch the modal dialog.

2. In the editor, select the button component on the page

3. On the Common page of the Property Inspector, expand the **Button Action** section.

4. In the **Action** field, enter an action, for example, `calltoTarget`.

The action must match the **from-outcome** on the control flow case leading to the task flow call activity. In [Figure 17-13](#), this is `calltoTarget`.

5. In the **UseWindow** drop-down list, select **true**.

Setting `useWindow` to **true** launches the target in a popup dialog.

6. In the source task flow diagram, select the task flow call activity.

7. In the Property Inspector, click **Behavior** and expand the **Run as Dialog** section.

8. In the **run-as-dialog** drop-down list, select **true**.

9. In the Application Navigator, double-click the page (`launch_page`) that will launch the modal dialog.

17.7.2 How to Pass Back a Return Value

The ADF bounded task flow can optionally return a value to the calling page when the modal dialog is dismissed. The return value can be displayed in an input UI component on the calling page.

The launching command button must specify a return listener. The return listener is a method binding for a method with one argument, a return event. The return listener takes the value specified in the `returnEvent` and sets it on the input component.

The input UI component should accept the return value of the command button by specifying a backing bean and setting its `partialTriggers` attribute to the id of the launching command button (see *Using Input Components and Defining Forms* in the *Oracle Fusion Middleware Web User Interface Developer's Guide for Oracle Application Development Framework* for more information). The backing bean containing the return listener should be registered in `adfc-config.xml`.

The target bounded task flow in the example contains a single view activity associated with a page that contains an output text component. The source task flow passes the value in the input text field as a parameter to the target task flow (for more information, see [Section 15.2, "Passing Parameters to an ADF Bounded Task Flow"](#)).

In addition, the target task flow should specify a return value definition. To return a value, you must specify:

- Return value definition on the called bounded task flow. This specifies where the return value is to be taken from upon exit of the called bounded task flow.
- Return values on the task flow call activity in the calling task flow. These specify where the calling task flow can find return values (for more information, see [Section 15.4, "Specifying Return Values"](#)).

Before you begin, follow the steps described in [Section 17.7.1, "How to Run an ADF Bounded Task Flow in a Modal Dialog"](#).

To specify a return value:

1. In the source task flow diagram, select the task flow call activity.
2. In the Property Inspector, click **Behavior** and expand the **Run as Dialog** section.
3. In the **run-as-dialog** drop-down list, select **true**.
4. In the Property Inspector, enter a **dialog-return-value**, for example, `returnvalue`.

The **dialog-return-value** must match the name of the return value definition you specified for the target bounded task flow. If you have not already specified a return value definition on the called task flow, see [Section 15.4, "Specifying Return Values"](#) for more information.

5. In the Application Navigator, double-click the page that will launch the modal dialog.
6. In the editor, select the Input UI component on the page.
7. In the Property Inspector, click **Behavior**.
8. In the PartialTriggers field, enter an EL expression for a partial trigger, for example, `# {pageFlowScope.backingBean.gotoModalDialog}`.

The input component on the launch_page can accept the return value of the command button by specifying a backing bean and setting its partialTriggers attribute to the id of the launching command button. In the example EL expression above, `backingBean` is the name of a backing bean and `gotoModalDialog` is the **id** of the launching command button shown in [Figure 17-14](#).

For information on how to create backing beans, see the *Using Input Components and Defining Forms* in the *Oracle ADF Faces Developer's Guide* for more information.

9. In the Property Inspector for the button UI component, click **Behavior**.
10. Expand the **Secondary Window**
11. In the **ReturnListener** field, enter an EL expression that specifies a reference to a return listener method in the page's backing bean, for example, `# {pageBean.listenerMethod}`.

The return listener method will be used to process a return event that is generated when the modal dialog is dismissed.

17.8 Creating an ADF Task Flow Template

You can create an ADF task flow template that other developers can use as a starting point when creating new ADF bounded task flows.

Note: You cannot use an ADF task flow template as the basis for creating a new ADF unbounded task flow.

The task flow template enables reuse because any ADF bounded task flow based on it has the same set of activities, control flows, input parameters, and managed bean definitions that the template contains. In addition, you can specify that changes to the template are automatically propagated to any task flow or template that is created based on the template.

For example, suppose you have set up an activity to be used as an exception handler for an ADF bounded task flow, such as a view activity associated with a page for

global exception handling. Or, the exception handler might be set up to handle exceptions typically expected to occur in a task flow. If you expect multiple ADF bounded task flows to rely on the same error handler, you might consider adding the error handler to a task flow template. New task flows created based on the template automatically have the error handling activity added to the template. See [Section 17.4, "Handling Exceptions"](#) for more information.

You can base an ADF task flow template on an existing ADF task flow template. You can also refactor an existing ADF bounded task flow to create a new task flow template (for more information, see [Section 13.7.3, "How to Convert ADF Bounded Task Flows"](#)).

If you select the **Update the Task Flow When the Template Changes** checkbox in the Create ADF Task Flow or Create ADF Task Flow Template dialog, the template will be reused by reference. Any changes you make to the template will be propagated to any ADF bounded task flow or ADF task flow template based on it.

17.8.1 Copying and Referencing an ADF Task Flow Template

There are two methods for reusing a ADF task flow templates.

- **Reuse by copy**

Deselect the **Update the task flow when the template changes** checkbox when creating a new ADF bounded task flow, as shown in [Figure 17-15](#). If you deselect the checkbox, the ADF bounded task flow is independent of the template. Changes to the template are not propagated to the ADF bounded task flow.

For example, if you add View activities associated with JSF pages to the template, the new ADF bounded task flow will display the Views, any control flows between them, and will retain the View associations with JSF pages. If you add a train on an ADF task flow template, any ADF bounded task flow created from it contains page navigation for the train.

- **Reuse by reference**

Select the **Update the task flow when the template changes** checkbox when creating a new ADF bounded task flow, as show in [Figure 17-15](#), or when creating a new ADF task flow template. Changes to the parent ADF task flow template propagate to any ADF bounded task flow or template based on it.

You can change, update, or disassociate the parent task flow template of a child ADF bounded task flow or task flow template at any point during development of the child.

At runtime, the contents of a child bounded task flow or a child template reused by reference are combined additively with the contents of the parent template. Any collision between the parent template and child task flow or child template are won by the child. For example, suppose you created a parent task flow template containing a train, and then created a child ADF bounded task flow based on the parent template. Later, you unchecked the **Train** checkbox on the Behavior page of the task flow definition Overview tab. The difference in how the Train checkbox is set for the parent template and the child task flow is a collision.

[Table 17-2](#) describes the specific combination algorithm used for each element. As shown in the table, in the event of a collision between train settings, the child task flow overrides the parent task flow template.

Table 17-2 Collision Resolution between Parent Template and Children

Task Flow Definition Metadata	Combination Algorithm
Default activity	Child bounded task flow or child task flow template overrides parent task flow template.
Transaction	Child bounded task flow or child task flow template overrides parent task flow template as an entire block of metadata, including all subordinate elements.
Task flow reentry	Child bounded task flow or child task flow template overrides parent task flow template, as an entire block of metadata, including all subordinate elements.
Control flow rules	Combination algorithm occurs at the control flow case level, not the control flow rule level. Control flow cases fall into the following categories: <ul style="list-style-type: none"> ▪ Both from action and from outcome specified ▪ Only from action specified ▪ Only from outcome specified ▪ Neither from action nor from outcome specified Each of these categories are merged additively. The child task flow definition or template overrides parent task flow template for identical matches within each of the four categories.
Input parameter definitions	Child bounded task flow or child task flow template overrides parent task flow template for identical input parameter definition names.
Return value definitions	Child bounded task flow or child task flow template overrides parent task flow template for identical return value definition names.
Activities	Child bounded task flow or child task flow template overrides parent task flow template for identical activity ids.
Managed beans	Child bounded task flow or child task flow template overrides parent task flow template for identical managed bean names.
Initializer	Child bounded task flow or child task flow template overrides parent task flow template.
Finalizer	Child bounded task flow or child task flow template overrides parent task flow template.
Critical	Child bounded task flow or child task flow template overrides parent task flow template.
Use page fragments	Child bounded task flow or child task flow template overrides parent task flow template.
Exception handler	Child bounded task flow or child task flow template overrides parent task flow template.
Security - permission	Child bounded task flow or child task flow template overrides parent task flow template.
Security - transport guarantee	Privilege maps are additive. Child bounded task flow or child task flow template overrides parent task flow template for identical privilege map operations.
Train	Child bounded task flow or child task flow template overrides parent task flow template.

Validations at both design time and runtime verify that the resulting parent - child extension hierarchy doesn't involve cycles of the same ADF task flow template.

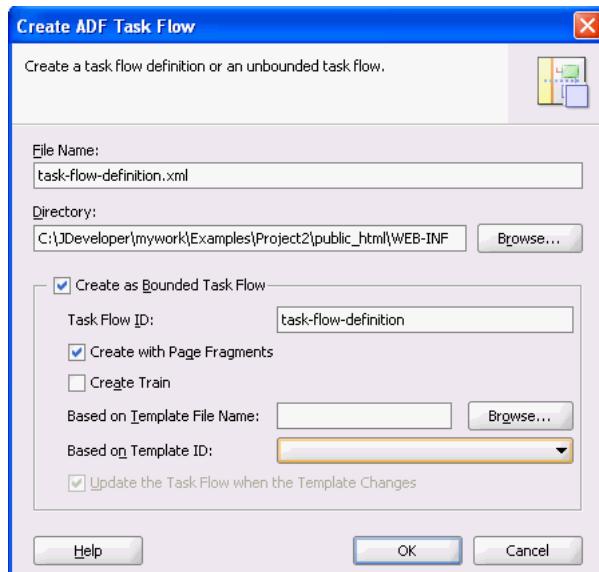
17.8.2 Creating an ADF Task Flow Template from Another Task Flow

The process for creating a new ADF task flow template from an existing template is similar to creating an ADF bounded task flow based on a template (see [Section 13.2, "Creating Task Flows"](#) for more information).

17.8.3 How to Use an ADF Task Flow Template

After you create an ADF task flow template, you can use it as the basis for creating a new ADF bounded task flow or a new task flow template. As shown in [Figure 17–15](#), the Create ADF Task Flow dialog has fields for creating an ADF bounded task flow based on the template file name and the id. You must specify both the file name and id of the template. These fields are available only if you select the **Create as Bounded Task Flow** checkbox.

Figure 17–15 Create ADF Task Flow Dialog



You can base a new template on an existing template. The Create ADF Task Flow Template dialog box contains fields for creating an ADF task flow template based on the file name and the template id of an existing task flow template.

You cannot run an ADF task flow template on its own. See [Section 13.5, "Running ADF Task Flows"](#) for more information.

17.8.4 How to Create an ADF Task Flow Template

The process for creating a new ADF task flow template is similar to creating an ADF bounded task flow. This section describes how to create a new ADF task flow template from scratch. You can also convert an existing ADF bounded task flow to a task flow template and vice versa (see [Section 17.8.2, "Creating an ADF Task Flow Template from Another Task Flow"](#) for more information).

To create an ADF task flow template:

1. In the Application Navigation, right-click the project where you want to create the task flow and choose **New**.
2. In the New Gallery, open the **Web Tier** node.
3. Click **JSF** and then click **ADF Task Flow Template**.
4. The value in **File Name** is used to name the XML source file for the ADF task flow template you are creating. The source file includes the activities and control flow rules that are in the task flow template. The default name for the XML source file is `task-flow-template.xml`.
5. In the Create ADF Task Flow Template dialog, the **Create with Page Fragments** checkbox is selected by default. If you expect that an ADF bounded task flow based on the template will be used as an ADF region, select this option.

If you want to add JSF pages instead of JSF page fragments to the task flow template, deselect the checkbox.

6. Click **OK**.

A diagram for the task flow template automatically opens in the editor.

7. You can add activities, control flows, and other items to the template.

Anything you can add to an ADF bounded task flow can be added to the ADF task flow template.

8. When you are finished, save your work.

The template will be available for use when you create an ADF bounded task flow or ADF task flow template.

17.8.5 What Happens When You Create an ADF Task Flow Template

As shown in [Example 17–5](#), an XML file is created each time you create a new ADF task flow template using JDeveloper. You can find the XML file in the Application Navigator in the location that you specified in the Directory field of the Create ADF Task Flow Template dialog, for example, `.../WEB-INF`.

The contents of the XML source file for the ADF task flow template can be similar to those of an ADF bounded task flow definition. One difference is the inclusion of the `<task-flow-template>` tag.

Example 17–5 ADF task flow template source file

```
<?xml version="1.0" encoding="windows-1252" ?>
<adfc-config xmlns="http://xmlns.oracle.com/adf/Controller">
  <task-flow-template id="task-flow-template">
    <default-activity>view1</default-activity>
    <view id="view1">view1.jsff</view>
  </task-flow-template>
</adfc-config>
```

17.8.6 What You Need to Know About ADF Task Flow Templates That Use Bindings

If you use an ADF task flow template that contains bindings, you must change the component IDs of ADF task flows based on the ADF task flow template. This ensures that the IDs are unique. ADF task flows generated from the template use the same IDs as the template ID. This may cause an exception at runtime.

For more information, see [Section 18.2.1, "How to Use ADF Data Binding in ADF Page Templates".](#)

17.9 Creating a Page Hierarchy

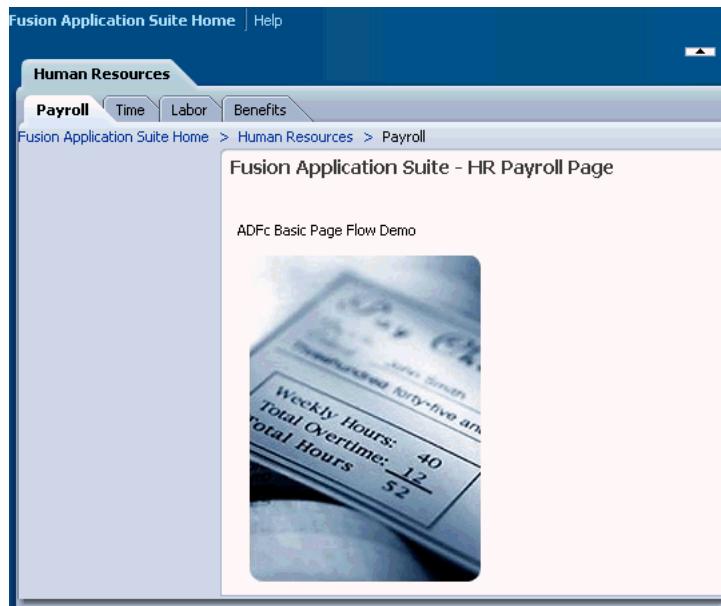
Note: This section describes creating a page hierarchy. If you follow the steps below, JDeveloper automatically generates the metadata required to create the hierarchy.

You also have the option of building the page hierarchy yourself using your own XML Menu model metadata. For more information, see the *Web User Interface Developer's Guide for Oracle Application Development Framework*.

The section in the developer's guide also contains detailed information about the XML menu model metadata.

You can create an application of related JSF pages that are organized in a tree-like hierarchy. End users access information on the pages by navigating a path of links. [Figure 17–16](#) shows a sample page hierarchy.

Figure 17–16 Page Hierarchy



To navigate, an end user clicks links on each page to drill down or up to another level of the hierarchy. For example, clicking **Human Resources** on the Application Suite Home page displays the page hierarchy shown in [Figure 17–16](#) above. Clicking the link on the **Benefits** tab displays the page hierarchy shown below in [Figure 17–17](#).

Figure 17-17 Benefits Page

The user can click links on the Benefits page to display other pages, for example, the Medical, Dental or Vision pages. The breadcrumbs on each page indicate where the current page fits in the hierarchy. The user can click each node in a breadcrumb to navigate to other pages in the hierarchy. The bold tab labels match the path to the current page shown the breadcrumbs.

You can use ADF Controller features in conjunction with XML Menu Model to build the page hierarchy. If you use this method, the following is generated automatically for you:

- Control flow metadata that determines which view/page will display when a menu item is selected.
- Default navigation widgets such as Previous and Next buttons
- Breadcrumbs
- XML menu model metadata
- Managed bean configuration

17.9.1 XML Menu Model

The XML menu model represents navigation for a page hierarchy in XML format. In the XML menu model metadata, a page hierarchy is described within the `menu` element, which is the root element of the file. Every XML menu model metadata file is required to have a `menu` element. Only one `menu` element is allowed.

A hierarchy is comprised of nodes. Each node corresponds to a page. The `menu` element can contain elements for the following nodes:

- `groupNode`: Contains one or more child nodes to navigate to indirectly. It must reference the id of at least one of its child nodes, which can be either another group node or an item node.
- `itemNode`: Specifies a node that performs navigation upon user selection.

- `sharedNode`: References another XML menu. A `sharedNode` is not a true node; it does not perform navigation nor does it render anything on its own. It is simply an include mechanism. Use a `sharedNode` to link or include various menus into a larger hierarchy.

The Benefits page in [Figure 17–17](#) above, for example, contains a group node called Benefits and three item nodes called Medical, Dental and Vision. The Benefits group node references its child Medical item node. Clicking on the Benefits menu item tab actually results in the Medical page being displayed.

When you create the pages in your hierarchy, you don't have to create any navigation links. For example, the Medical, Dental and Vision links are automatically built on the Benefits page if you make the first the three links item nodes and Benefits a group node.

17.9.2 How to Create a Page Hierarchy

You also don't need to create the nodes in the metadata file for your page hierarchy. JDeveloper does that for you. All you need to do is organize the nodes in the hierarchy.

To create a page hierarchy, you first divide the nodes into menus. [Figure 17–18](#) shows the division of the Human Resources page hierarchy into menus:

Note: It is possible to create the entire menu hierarchy in one menu model. However, breaking a menu hierarchy into submenus makes maintenance easier. In addition, breaking the menu hierarchy into smaller submenu models enables each separate development organization to develop its own menu. These separate menus can later be combined using shared nodes to create the complete menu hierarchy.

- The top level menu contains the Fusion Application Suite Home page. This is the root parent page and contains a single tab that links to the Human Resources sub menu model.

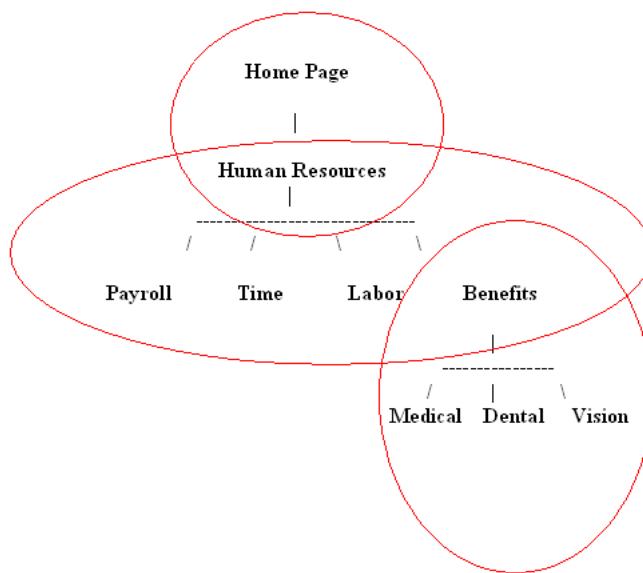
In this menu, the Home page is represented as an item node and the Human Resources page as a shared node.

- The Human Resources' four tabs link to child Payroll, Time, Labor, and Benefits pages.

In this menu, Human Resources is a group node referring to its child Payroll item node, two additional item nodes for the Time and Labor pages, and a Benefits shared node that includes the Benefits submenu model.

- The Benefits node is a page that contains links to the Medical, Dental and Vision pages.

In this menu, Benefits page is a group node referring to its child Medical item node, and two additional item nodes for the Dental and Vision pages.

Figure 17–18 Menu Hierarchy**To create a page hierarchy:**

1. Create the pages you will use in your menu hierarchy. See [Section 18.3, "Creating a Web Page"](#) for more information.

Create one page for each node in the hierarchy.

Note: You are not required to create your pages first. Instead, you can wait until later in the process, for example, after you've organized the page hierarchy following the steps below.

2. For each menu that will be in the final page hierarchy create an XML source files by deselecting the **Create as Bounded Task Flow** checkbox in the Create ADF Task Flow dialog. JDeveloper will combine these XML source files into a single ADF unbounded task flow. See [Section 13.2.1, "How to Create an ADF Task Flow"](#) for more information.

For example, [Figure 17–18](#) contains three menus:

- a top level or root menu that contains the Fusion Application Suite Home page.
- the Human Resources menu
- the Benefits menu

Therefore you should create three XML source files, for example:

- adfc-config.xml source file (always) corresponds to the root menu
- adfc-hr-config.xml source file corresponds to the Human resources menu
- adfc-benefits-config.xml source file corresponds to the Benefits menu

Tip: Prefix all of your source file names for the unbounded task flows with `adfc_`. This helps to identify the file as the source for an unbounded task flow, as opposed to a bounded task flow definition.

3. In the Application Navigator, double-click each source file to open it in the editor.
4. Add view activities to each source file that correspond to each menu node in the appropriate menu.

For example, the Benefits menu will contain one group node (`benefits`) and three item nodes (`medical`, `dental` and `vision`). Therefore, add four view activities to the source for the Benefits menu.



Note: For menus that will include other menus using shared nodes, do not create views for the shared nodes. For example the HR menu will have a node at the end called Benefits that will include the Benefits menu using a shared node. Because a shared node is not really a node, there is no need to create a view for the Benefits page. This view has already been put in the XML source file associated with Benefits, `adfc-benefits-config.xml`.

Similarly, the Fusion Application Home Page menu includes the Human Resources menu at the end by using a shared node. Therefore, there is no need to create a view for the Human Resources page in `adfc-config.xml`. It is already in the XML source file associated with Human Resources, `adfc-hr-config.xml`.

5. Associate the pages you created for the hierarchy with the views.

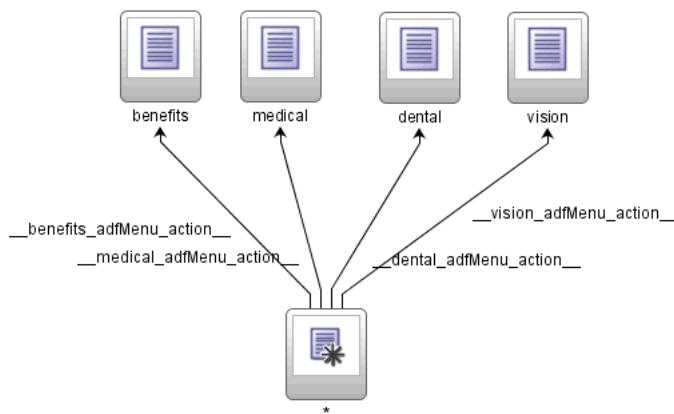
The easiest way to do this is to drag the page from the Application Navigation and drop it onto its corresponding view activity in the task flow diagram.

6. In the Application Navigator, right-click each XML source file included in the page hierarchy and choose **Create ADF Menu**.

For example, you could begin by right-clicking on `adfc_benefits.xml`. This bottom up method is the best for this scenario because the parent menus include the submenus (via shared nodes). It is better if the submenus are created first.

7. In the Create ADF Menu Model dialog, enter a **File Name** for the menu XML file that will be generated and a **Directory** where it will be located.
8. Click **OK**.

The XML source file is updated with the control flows and managed beans used to navigate. In general, you don't have to change anything in this source file. However, you can use ADF Controller features to update any of this information. You might change the **from-outcome** default values (such as `_benefits_adfMenu_action_`) given to each of the control flow cases in the diagram.

Figure 17–19 Updated Unbounded Task Flow Diagram

In addition, a new menu XML file is created. Keep in mind that the menu XML file is different from the XML source file for the unbounded task flow.

9. In the Application Navigator, scroll down to the menu XML file and select it.

The file will be in the Application Navigator at the location you selected in the **Directory** field of the Create ADF Menu Model dialog.

In the Structure window, all of the views that are in the unbounded task flow now have corresponding item nodes in the menu XML file. All item nodes are at the same level. They are all siblings and none is a parent of any other node by default.

10. Create a hierarchy of group and child nodes.

The group node must refer to at least one child node. For example, before converting the Benefits node from an item node to a group node, you must identify its child nodes (Medical, Dental, and Vision).

- a. In the Structure window, drag and drop each child menu onto its parent.
- b. Right-click on the parent node (for example, Benefits) and choose **Convert**.
- c. In the Convert dialog choose **groupNode** and click **OK**.
- d. In the **id** field, enter a new identifier or accept the default.

The **id** must be unique among all of the nodes in all of the menu models. It is good practice to specify an id that identifies the node. For example, if you change the Benefits node to a group node, you can update its default id, `_benefits_itemNode_`, to `_benefits_groupNode_`.

- e. In the **idref** field, enter the id of one of the other nodes in the menu, for example, `_medical_itemNode_`.
The value you enter can be an id of a child that is a group node or an item node.
- f. Enter or change the existing default value in the **label** field to match what you want to display. For example, you might change `_benefits_label_` to `Benefits`.
- g. Accept the rest of the default values in the fields and click **Finish**.

- h. The Confirm Convert dialog asks if you want to delete the `action` and `focusViewID` attributes on the `groupNode` element. Since group nodes do not use these attributes, always click **OK**.

11. If you selected **sharedNode** on the Convert dialog:

 - a. In the `ref` field, enter an EL expression that references another XML menu file, for example, `# {benefits_menu}`.

`benefits_menu` is a `<managed-bean-name>` property of the managed bean that defines the menu model for the Benefits menu. You can find this managed at the bottom of the XML source file for the `benefits_menu`.

For example, the HR menu contains four item nodes. One of the item nodes, Benefits, should include another menu called `benefits_menu`. In the Structure window for the HR menu file, you would select the Benefits item node and choose **Convert > sharedNode**. In the `ref` field, you could enter `# {benefits_menu}`.

b. Click **OK**.

Now you have included the HR menu into the root menu.

 - c. Update all of your menu nodes in all menu XML files. The default generated labels are probably not what you want in your application.

12. You can also add a new node to a menu instead of converting an existing one.

 - a. In the Structure window scroll to the location where you want add the node.
 - b. Right click and choose an **Insert** option.
 - c. Choose **itemNode**, **groupNode** or **sharedNode**.

13. To run the finished page hierarchy:

 - a. In the Application Navigator, right-click on your project node and choose **Build Project Working Set**.
 - b. In the Application, Navigator, scroll to the unbounded task flow containing the root menu. This should always be called `adfc_config.xml`.
 - c. If everything compiles without errors, right-click and choose **Run**. For more information, see [Section 13.5.5, "How to Run an ADF Unbounded Task Flow"](#).

Note: If your page hierarchy is based on more than one unbounded task flow in the same project, ensure that each additional unbounded task flow configuration file is listed as a `<metadata-resources>` element in the top-level `adfc_config.xml` file for the hierarchy. For more information, see [Section 17.10, "How to Specify Bootstrap Configuration Files"](#).

17.10 How to Specify Bootstrap Configuration Files

The top level XML source files in an application are known as the application's bootstrap configuration files. The content and loading mechanism for each file mimics `faces-config.xml`, the XML source file used in JSF navigation. When you create a new ADF unbounded task flow, JDeveloper automatically adds an entry into `adfc-config.xml` to include the task flow it in the bootstrap list.

Bootstrap XML source files are loaded when an application is first started. The files can contain:

- ADF navigation metadata for an ADF unbounded task flow
- ADF activity metadata for an ADF unbounded task flow
- Managed bean definitions used by ADF activities

The main bootstrap XML source file is usually named `adfc-config.xml`. A single web application can have multiple top level XML source files, which are loaded when application is first started. Although the bootstrap XML source files can be implemented as separate physical files, they are all considered the same logical file at runtime.

You can add additional bootstrap XML source files in the Metadata Resources list. To access the list, click the **Overview** tab for `adfc-config.xml` or some other bootstrap configuration file, then click **Metadata Resources**.

Note: It is a best practice to specify metadata resources within the `adfc-config.xml` file or any of the bootstrap configuration files referenced within `adfc-config.xml`.

17.11 Using the `adf-config.xml` File to Configure ADF Controller

The central configuration file for all of ADF is `adf-config.xml`. This file contains sections that configure different ADF components, for example, ADF Controller behavior such as save for later.

17.11.1 Specifying Save for Later Settings

You can specify save for later settings in the `adf-config.xml` file such as whether implicit save points can be created in the application, as shown in [Table 17–3](#).

Table 17–3 Save For Later Configuration Properties

Property	Description
<code>savepoint-datasource</code>	JNDI name of the data source, for example, <code>java:comp/env/jdbc/Connection1DS</code> The value for this property is different from the JDeveloper Database Connection name. JDeveloper datasources typically append a "DS" on the end of the database connection name.
<code>enable-implicit-savepoints</code>	When set to <code>true</code> , implicit save points can be created for the application. See Section 17.5.1, "How to Add Save For Later Capabilities" for information about creating a save point. The default setting for this property is <code>false</code> .

17.11.2 Validating ADF Controller Metadata

Basic validation is performed when ADF Controller retrieves metadata. The most serious errors, for example, a task flow that is missing a default activity, result in parsing exceptions.

The `enable-grammar-validation` setting in `adf-config.xml` allows you to validate the grammar in ADF Controller metadata before deploying an application. When `enable-grammar-validation` is set to `true`, ADF Controller metadata is validated against ADF Controller XSDs. For example, invalid characters in ADF Controller metadata such as a slash (/) in a view activity id are flagged as exceptions.

By default, enable-grammar-validation is set to false. For performance reasons, it should be set to true only during application development or when troubleshooting an application.

17.11.3 Searching for adf-config.xml

ADFConfigFactory first searches for the adf_config.xml file in META-INF. The first file it finds in this location is used.

The integrated development environment (IDE) creates adf-config.xml in the location <application root>/.adf/META-INF/adf-config.xml. By default, when the application runs or a .war file is built, this location is included. However, if you create an adf-config.xml file and put it in <project root>/META-INF/adf-config.xml, you cannot guarantee it will be used. If there is another adf-config.xml, then it will be used instead.

The adf-config.xml file in WEB-INF is parsed but only if the filter oracle.adf.share.http.ServletADFFilter is installed. This adf-config.xml file is merged on top of the one found in META-INF. In this way, it is possible to have both an application-level adf-config.xml file and a project-level adf-config.xml configuration file.

17.11.4 Replicating Memory Scopes in a Server-Cluster Environment

Typically, in an application that runs in a clustered environment, a portion of the application's state is serialized and copied to another server or a data store at the end of each request so that the state is available to other servers in the cluster.

When you are designing an application to run in a clustered environment, you must:

- Ensure all managed beans with a lifespan longer than one request are serializable (that is, they implement the java.io.Serializable interface). Specifically, beans stored in sessionScope, pageFlowScope, and viewScope need to be serializable.
- Make sure that the ADF framework is aware of changes to managed beans stored in ADF scopes (viewScope and pageFlowScope).

When a value within a managed bean in either viewScope or pageFlowScope is modified, the application needs to notify ADF framework so it can ensure the bean's new value get replicated.

In [Example 17–6](#), an attribute of an object in viewScope is being modified.

Example 17–6 Code that Modifies an Object in viewScope

```
Map<String, Object> viewScope =  
    AdfFacesContext.getCurrentInstance().getViewScope();  
MyObject obj = (MyObject)viewScope.get("myObjectName");  
Obj.setFoo("newValue");
```

Without additional code, the ADF framework will be unaware of this change and will not know that a new value needs to be replicated within the cluster. To inform the ADF framework that an object in an ADF scope has been modified and that replication is needed, use the markScopeDirty() method similar to what is shown in [Example 17–7](#). The markScopeDirty() method accepts only viewScope and pageFlowScope as parameters.

Example 17–7 Additional Code to Notify the ADF Framework of Changes to an Object

```
ControllerContext ctx = ControllerContext.getInstance();
ctx.markScopeDirty(viewScope);
```

This code is needed for any request that modifies an existing object in one of the ADF scopes. It is not necessary to notify the ADF framework if the scope itself is modified, such as by calling the scope's `put()`, `remove()`, or `clear()` methods.

If an application is not deployed to a clustered environment, the tracking of changes to ADF memory scopes is not needed, and by default, this functionality is disabled. To enable the ADF Controller to track changes to ADF memory scopes and replicate the pageFlowScope and viewScope within the server cluster, set the

`<adf-scope-ha-support>` parameter in the `adf-config.xml` file to `true`. Because scope replication has a small performance overhead, it should be enabled only for applications running in a server-cluster environment.

Example 17–8 shows `adf-scope-ha-support` set to `true` in the `adf-config.xml` file.

Example 17–8 *adf-scope-ha-support* Parameter in the *adf-config.xml* File

```
<adf-scope-ha-support>true</adf-scope-ha-support>
```


Part IV

Creating a Databound Web User Interface

Part IV contains the following chapters:

- [Chapter 18, "Getting Started with Your Web Interface"](#)
- [Chapter 19, "Understanding the Fusion Page Lifecycle"](#)
- [Chapter 20, "Creating a Basic Databound Page"](#)
- [Chapter 21, "Creating ADF Databound Tables"](#)
- [Chapter 22, "Displaying Master-Detail Data"](#)
- [Chapter 23, "Creating Databound Selection Lists and Shuttles"](#)
- [Chapter 24, "Creating Databound ADF Data Visualization Components"](#)
- [Chapter 25, "Creating ADF Databound Search Forms"](#)
- [Chapter 26, "Creating More Complex Pages"](#)
- [Chapter 27, "Designing a Page Using Placeholder Data Controls"](#)

Getting Started with Your Web Interface

This chapter describes the steps you may need to take before using the Data Controls panel to create databound UI components on JSF pages.

The chapter includes the following sections:

- [Section 18.1, "Introduction to Developing a Web Application with ADF Faces"](#)
- [Section 18.2, "Using Page Templates"](#)
- [Section 18.3, "Creating a Web Page"](#)
- [Section 18.4, "Using a Managed Bean in a Fusion Web Application"](#)

18.1 Introduction to Developing a Web Application with ADF Faces

Most of what you need to know to get started with your web interface is covered in the *Oracle Fusion Middleware Web User Interface Developer's Guide for Oracle Application Development Framework*. However, using the ADF Model layer for data binding instead of JSF managed beans provides additional functionality, such as the ability to declaratively bind components to your business services (for more information on what ADF Model can provide, see [Section 1.2.2, "ADF Model Layer"](#)). This chapter provides a high-level overview of the web interface development process as detailed in the Faces guide, and also provides information about the additional functionality when using ADF Model data binding.

Following the development process outlined in [Chapter 1, "Introduction to Building Fusion Web Applications with Oracle ADF"](#), developing a web application with ADF Faces and using ADF Model for data binding involves the following steps:

- Creating ADF Faces templates for your pages (optional)
- Creating the individual pages and page fragments for regions to be used within a page
- Creating any needed managed beans

Additionally, the lifecycle of a Fusion web application is different from that of a standard JSF or ADF Faces application. For more information about how the lifecycle works, see [Chapter 19, "Understanding the Fusion Page Lifecycle"](#).

18.2 Using Page Templates

As you design the flow of your application, you can begin to think about the design of your pages. To ensure consistency throughout your application, you use ADF page templates. These templates provide structure and consistency for other developers building web pages. Templates can have both static areas that cannot be changed

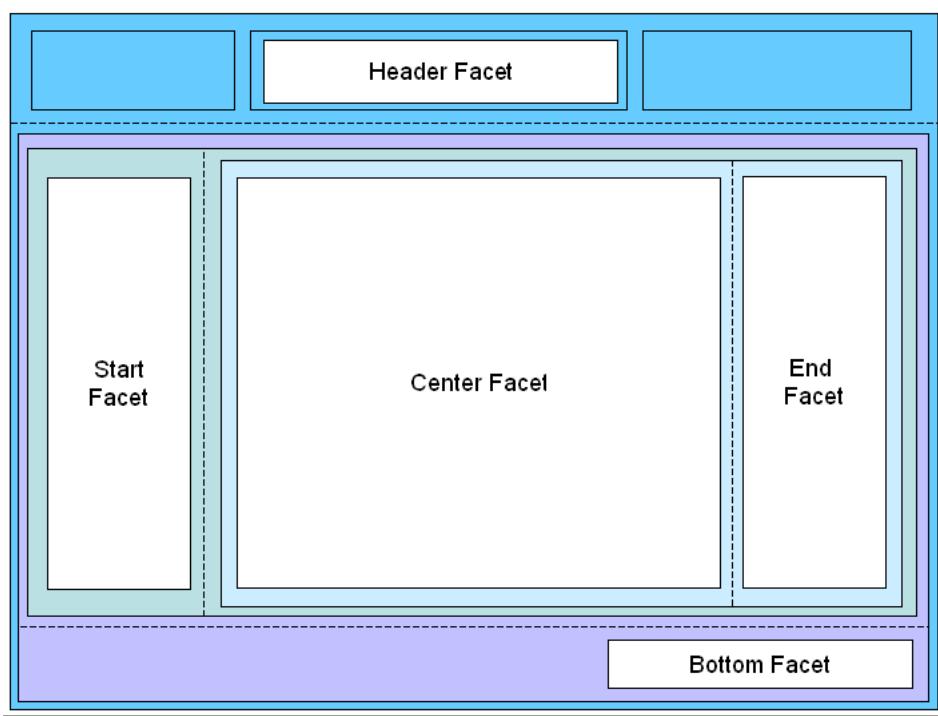
when they are used, and dynamic areas where the developer can place content specific to the page they are building.

For example the StoreFront module of the Fusion Order Demo application contains a template that provides a top area for branding and navigation, a bottom area for copyright information, and a center area for the main content of the page. Page developers do not need to do anything to the branding and copyright information when they use the template. They need only to develop the main content area.

In addition to using ADF Faces components to build a template, you can add attributes to the template definition. These attributes provide placeholders for specific information that the page developer needs to provide when using the template. For example, the template in the StoreFront module application contains an attribute for a welcome message. When a page developers use this template, they can add a different welcome message for each page.

You can also add facet references to the template. These references act as placeholders for content on the page. [Figure 18–1](#) shows a rendition of how the `StoreFrontTemplate` template used in the StoreFront module application uses facet references.

Figure 18–1 Facets in the `StoreFrontTemplate`



In this template, facet references are used inside four different `panelSplitter` components. When the home page was created using this template, the navigational links were placed in the Header facet, the accordion panels that hold the navigation trees and search panels were placed in the Start facet. The cart summary was placed in the End facet, and the main portion of the page was placed in the Center facet. The copyright information was placed in the Bottom facet.

When you choose to add databound components to a page template, an associated page definition file and the other metadata files that support ADF Model layer data binding are created. Each component is bound in the same fashion as for standard JSF pages, as described in [Chapter 11, "Using Oracle ADF Model in a Fusion Web](#)

[Application](#)". You can also create model parameters for the template. The values for these parameter can then be set at runtime by the calling page.

For example, if you wanted a list of products to appear on each page that uses the template, you could drag and drop the `ProductName` attribute of the `Products` collection as a list. Or, if you wanted the pages to display the currently selected product ID, you could create a model parameter for the template that would evaluate to that product's ID.

Note: Templates are primarily a project artifact. While they can be reused between projects and applications using an ADF Library, they are not fully self-contained and will always have some dependencies to external resources, for example, any ADF data binding, Strings from a message bundle, images, and managed beans.

18.2.1 How to Use ADF Data Binding in ADF Page Templates

Creating a page template for use in an application that uses ADF Business Components and ADF Model layer data binding is the same as for creating a standard ADF Faces page template, as documented in the "Creating Page Templates" section in the *Oracle Fusion Middleware Web User Interface Developer's Guide for Oracle Application Development Framework*. Once you create the template, you can drag and drop items from the Data Controls panel. JDeveloper automatically adds the page definition file when you drag and drop items from the Data Controls panel.

The Create JSF Page Template wizard also allows you to create model parameters for use by the template.

To add model parameters to a template:

1. Create a page template following the instructions in the "How to Create a Page Template" section in the *Oracle Fusion Middleware Web User Interface Developer's Guide for Oracle Application Development Framework*. However, do not complete the wizard.
2. In the Create JSF Page Template wizard, select **Create Associated ADFm Page Definition**.

Note: You only need to select this checkbox if you wish to add parameters. JDeveloper automatically adds the page definition file when you drag and drop items from the Data Controls panel.

3. In the JSF Page Template wizard, click the **Model Parameters** tab.
4. Click the **Add** icon.
5. Enter the following for the parameter:
 - **Id:** Enter a name for the parameter.
 - **Value:** Enter a value or an EL expression that can resolve to a value. If needed, click the ellipses button to open the Expression Builder. You can use this to build the expression. For more information about EL expressions and the EL expression builder, see [Chapter 11.7, "Creating ADF Data Binding EL Expressions"](#).
 - **Option:** This determines how the parameter value will be specified. Enter `optional` if the value does not need to be specified. Enter `final` if the

parameter value cannot be supplied from the usage, and must have some default value or other means of obtaining the value, for example from an EL expression. Enter **mandatory** if the value must be specified by the usage.

- **Read Only:** Select if the parameter's value is to be read-only and should not be overwritten
6. Create more parameters as needed. Use the order buttons to arrange the parameters into the order in which you want them to be evaluated.

You can now use the Data Controls panel to add databound UI components to the page, as you would for a standard JSF page, as described in the remaining chapters in this part of the book.

Note: If your template contains any method actions bound to a method iterator, you cannot change the value of the `refresh` attribute on the iterator to anything other than `Default`. If set to anything other than `Default`, the method will not execute.

18.2.2 What Happens When You Use ADF Model Layer Bindings on a Page Template

When you add ADF databound components to a template or you create model parameters for a template, a page definition file is created for the template, and any model parameters are added to that file. [Example 18–1](#) shows what the page definition file for a template for which you created a `productId` model parameter might look like.

Example 18–1 Model Parameters in a Page Definition for a Template

```
<parameters>
  <parameter id="productId" readonly="true"
             value="#{bindings.productId.inputValue}" />
</parameters>
<executables/>
<bindings/>
```

Parameter binding objects declare the parameters that the page evaluates at the beginning of a request (for more information about binding objects and the ADF lifecycle, see [Chapter 11, "Using Oracle ADF Model in a Fusion Web Application"](#)). However, since a template itself is never executed, the page that uses the template (the *calling page*) must access the binding objects created for the template (including parameters or any other type of binding objects created by dragging and dropping objects from the Data Controls panel onto the template).

In order to access the template's binding objects, the page definition file for the calling page must instantiate the binding container represented by the template's page definition file. As a result, a reference to the template's page definition is inserted as an executable into the calling page's page definition file, as shown in [Example 18–2](#).

Example 18–2 Reference to Templates Page Definition as an Executable

```
<executables>
  <page path="oracle.foddemo.storefront.pageDefs.templates_MyTemplatePageDef"
        id="pageTemplateBinding" />
</executables>
```

In this example, the calling page was built using the `MyTemplate` template. By placing the page definition file for the `MyTemplate` template as an executable for the

calling page, when the calling page's binding container is instantiated, it will in turn instantiate the `MyTemplatePageDef`'s binding container, thus allowing access to the parameter values, or any other databound values.

By providing an ID for this reference (in this case, `pageTemplateBinding`), the calling page can have components that are able to access values from the template. When you create a JSF page from a template, instead of repeating the code contained within the template, the calling page uses the `af:pageTemplate` tag. This tag contains the path to the template JSF page.

Additionally, when the template contains any ADF data binding, the value of that tag is the ID set for the calling page's reference to the template's page definition, as shown in [Example 18–3](#). This binding allows the component access to the binding values from the template.

Example 18–3 Page Template Page Definition Reference

```
<af:pageTemplate viewId="/MyTemplate.jspx"
                 value="#{bindings.pageTemplateBinding}"/>
```

18.2.3 What Happens at Runtime: How Pages Use Templates

When a page is created using a template that contains ADF data binding, the following happens:

1. The calling page follows the standard JSF/ADF lifecycle, as documented in [Chapter 19, "Understanding the Fusion Page Lifecycle"](#). As the page enters the Restore View phase, the URL for the calling page is sent to the binding context, which finds the corresponding page definition file.
2. During the Initialize Context phase, the binding container for the calling page is created based on the page definition file.
3. During the Prepare Model phase, the page template executable is refreshed. At this point, the binding container for the template is created based on the template's page definition file, and added to the binding context.
4. The lifecycle continues, with UI components and bindings from both the page and the template being processed.

18.3 Creating a Web Page

Creating a web page for an application that uses ADF Model layer data binding is no different than described in the "Creating a JSF Page" section in the *Oracle Fusion Middleware Web User Interface Developer's Guide for Oracle Application Development Framework*. You can create pages either by double-clicking a view activity in a task flow or by using the New Gallery. When creating the page (or dropping a view activity onto a task flow), you can choose to create the page as a JSF JSP or as a JSF JSP fragment. JSF fragments provide a simple way to create reusable page content in a project, and are what you use when you wish to use task flows as regions on a page. When you modify a JSF page fragment, the JSF pages that consume the page fragment are automatically updated.

When you begin adding content to your page, you typically use the Component Palette and Data Controls panel of JDeveloper. The Component Palette contains all the ADF Faces components needed to declaratively design your page. Once you have the basic layout components placed on the page, you can then drag and drop items from the Data Controls panel to create ADF Model databound UI components. The

remaining chapters in this part of the book explain in detail the different types of data bound components and pages you can create using ADF Model data binding.

18.4 Using a Managed Bean in a Fusion Web Application

Managed beans are Java classes that you register with the application using various configuration files. When the JSF application starts up, it parses these configuration files, and the beans listed within them are made available. The managed beans can be referenced in an EL expression, allowing access to the beans' properties and methods. Whenever a managed bean is referenced for the first time and it does not already exist, the Managed Bean Creation Facility instantiates the bean by calling the default constructor method on it. If any properties are also declared, they are populated with the declared default values.

Often, managed beans handle events or some manipulation of data that is best handled at the front-end. For a more complete description of how managed beans are used in a standard JSF application, see the Java EE 5 tutorial on Sun's web site (<http://java.sun.com>)

Best Practice: Use managed beans to only store logic that is related to the UI rendering. All application data and processing should be handled by logic in the business layer of the application. Similar to how you store data-related logic in the database using PLSQL rather than a Java class, the rule of thumb in a Fusion web application is to store business-related logic in the middle tier. This way, you can expose this logic as business service methods, which can then become accessible to the ADF Model layer and be available for data binding.

In an application that uses ADF data binding and ADF task flows, managed beans are registered in different configuration files from those used for a standard JSF application. In a standard JSF application, managed beans are registered in the `faces-config.xml` configuration file. In a Fusion web application, managed beans can be registered in either the `faces-config.xml` file, the `adfc-config.xml` file, or a task flow definition file. Which configuration file you use to register a managed bean depends on what will need to access that bean, whether or not it needs to be customized at runtime, what the bean's scope is, and in what order all the beans in the application need to be instantiated. Table 18-1 describes the how registering a bean in each type of configuration file affects the bean.

Note: Placing managed beans within the `faces-config.xml` file is not recommended in a Fusion web application.

Managed beans accessed within the task flow definition must be registered in that task flow's definition file.

Table 18–1 Effects of Managed Bean Configuration Placement

Managed Bean Placement	Effect
adfc-config.xml	<ul style="list-style-type: none"> ▪ Managed bean can be of any scope. However, any backing beans for page fragments or declarative components should use BackingBean scope. For more information regarding scope, see Section 19.3, "Object Scope Lifecycles". ▪ When executing within an unbounded task flow, faces-config.xml will be checked for managed bean definitions before the adfc-config.xml file. ▪ Lookup precedence is enforced per scope. Request-scoped managed beans take precedence over session-scoped managed beans. Therefore, a request-scoped managed bean named <code>foo</code> in the adfc-config.xml file will take precedence over a session-scoped managed bean named <code>foo</code> in the current task flow definition file. ▪ Already instantiated beans take precedence over instantiating new instances. Therefore, an existing session-scoped managed bean named <code>foo</code> will always take precedence over a request-scoped bean named <code>foo</code> defined in the current task flow definition file.
Task flow definition file	<ul style="list-style-type: none"> ▪ Managed bean can be of any scope. However, managed beans of request, pageFlow, or with the scope set to none that are to be accessed within the task flow definition must be defined within the task flow definition file. Any backing beans for page fragments in a task flow should use BackingBean scope. ▪ Managed bean definitions within task flow definition files will be visible only to activities executing within the same task flow. ▪ When executing within a bounded task flow, faces-config.xml will be checked for managed bean definitions before the currently executing task flow definition. If no match is found in either location, adfc-config.xml and other bootstrap configuration files will be consulted. However, this lookup in other adfc-config.xml and bootstrap configuration files will only occur for session- or application-scoped managed beans. ▪ Lookup precedence is enforced per scope. Request-scoped managed beans take precedence over session-scoped managed beans. Therefore, a request-scoped managed bean named <code>foo</code> in the adfc-config.xml file will take precedence over a session-scoped managed bean named <code>foo</code> in the current task flow definition file. ▪ Already instantiated beans take precedence over instantiating new instances. Therefore, an existing session-scoped managed bean named <code>foo</code> will always take precedence over a request-scoped bean named <code>foo</code> registered in the current task flow definition file.
faces-config.xml	<ul style="list-style-type: none"> ▪ Managed beans can be of any scope other than page flow scope. ▪ When searching for any managed bean, the faces-config.xml file is always consulted first. Other configuration files will be searched only if a match is not found. Therefore, beans registered in the faces-config.xml file will always win any naming conflict resolution.

As a general rule for Fusion web applications, a bean that may be used in more than one page or task flow, or that is used by pages within the main unbounded task flow (`adfc-config`), should be registered in the `adfc-config.xml` configuration file. A managed bean that will be used only by a specific task flow should be registered in that task flow's definition file. There should be no beans registered in the `faces-config.xml` file.

Note: If you create managed beans from dialogs within JDeveloper, the bean is registered in the `adf-config.xml` file, if it exists.

For example, in the StoreFront module, the `myOrdersBean` managed bean is used by the `myOrders.jsp` page to handle the case where a user decides to cancel editing an order, and the edits have already been committed to the model but have not yet been persisted to the database. Because this bean is used by a page within the `adfc-config` unbounded task flow, it is registered in the `adfc-config.xml` file. The `welcomeUserRegistrationBean` is a managed bean used by the `welcomeUserRegistration` JSF fragment to handle the choice made by the user on that page. Because it is used solely within the `user-registration` task flow, it is registered in the `user-registration-task-flow` definition file.

This section describes how to create a managed bean for use within a task flow (either the default `adfc-config` flow or any bounded task flow). For more information regarding managed beans and how they are used as backing beans for JSF pages, see the "Creating and Using Managed Beans" section in the *Oracle Fusion Middleware Web User Interface Developer's Guide for Oracle Application Development Framework*

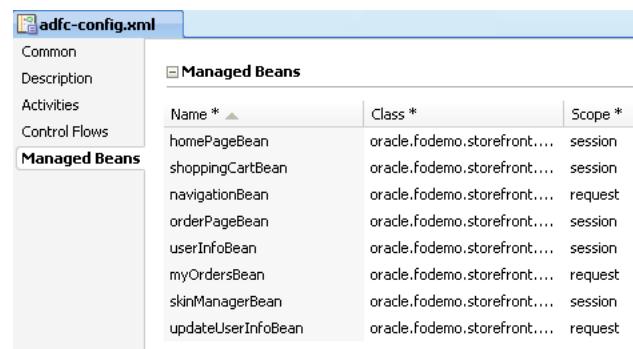
18.4.1 How to Use a Managed Bean to Store Information

Within the editors for a task flow definition, you can create a managed bean and register it with the JSF application at the same time.

To create a managed bean for a task flow:

1. In the Application Navigator, open either the `adfc-config.xml` file or any other task flow definition file.
2. At the bottom of the window, select the **Overview** tab.
3. In the navigation list on the left, select the **Managed Beans** tab. [Figure 18–2](#) shows the editor for the `adfc-config.xml` file.

Figure 18–2 Managed Beans in the `adfc-config.xml` File



Name *	Class *	Scope *
homePageBean	oracle.fodemo.storefront....	session
shoppingCartBean	oracle.fodemo.storefront....	session
navigationBean	oracle.fodemo.storefront....	request
orderPageBean	oracle.fodemo.storefront....	session
userInfoBean	oracle.fodemo.storefront....	session
myOrdersBean	oracle.fodemo.storefront....	request
skinManagerBean	oracle.fodemo.storefront....	session
updateUserInfoBean	oracle.fodemo.storefront....	request

4. Click the **Add** icon to add a row to the **Managed Beans** table.

5. In the Property Inspector, enter the following:

- **managed-bean-name**: A name for the bean.
- **managed-bean-class**: If the corresponding class has already been created for the bean, use the browse (...) button for the **managed-bean-class** field to search for and select the class. If a class does not exist, enter the name you'd like to use. Be sure to include any package names as well.
- **managed-bean-scope**: The bean's scope. For more information about scope values, see [Section 19.3, "Object Scope Lifecycles"](#).

Note: While not technically a scope, you can set the scope to none, meaning that the bean will not live within any particular scope, but will instead be instantiated each time it is referenced. You should set a bean's scope to none when it is referenced by another bean.

6. You can optionally add needed properties for the bean. With the bean selected in the **Managed Beans** table, click the **Add** icon for the **Managed Properties** table. In the Property Inspector, enter a property name (other fields are optional).

Note: While you can declare managed properties using this editor, the corresponding code is not generated on the Java class. You will need to add that code by creating private member fields of the appropriate type and then using the **Generate Accessors** menu item on the context menu of the code editor to generate the corresponding getter and setter methods for these bean properties.

18.4.2 What Happens When You Create a Managed Bean

When you use the configuration editor to create a managed bean and elect to generate the Java file, JDeveloper creates a stub class with the given name and a default constructor. [Example 18–4](#) shows the code added to the MyBean class stored in the view package.

Example 18–4 Generated Code for a Managed Bean

```
package view;

public class MyBean {
    public MyBean() {
    }
}
```

You now need to add the logic required by your task flow or page. You can then refer to that logic using an EL expression that refers to the managed-bean-name value given to the managed bean. For example, to access the myInfo property on the bean, the EL expression would be:

```
#{my_beans.myInfo}
```

JDeveloper also adds a managed-bean element to the appropriate task definition file. [Example 18–5](#) shows the managed-bean element created for the MyBean class.

Example 18–5 Managed Bean Configuration on the adfc-config.xml File

```
<managed-bean>
  <managed-bean-name>my_beans</managed-bean-name>
  <managed-bean-class>view.MyBean</managed-bean-class>
  <managed-bean-scope>session</managed-bean-scope>
</managed-bean>
```

Understanding the Fusion Page Lifecycle

This chapter describes the ADF page lifecycle, its phases, and how to best use the lifecycle within a Fusion web application.

This chapter includes the following sections:

- [Section 19.1, "Introduction to the Fusion Page Lifecycle"](#)
- [Section 19.2, "The JSF and ADF Page Lifecycles"](#)
- [Section 19.3, "Object Scope Lifecycles"](#)
- [Section 19.4, "Customizing the ADF Page Lifecycle"](#)

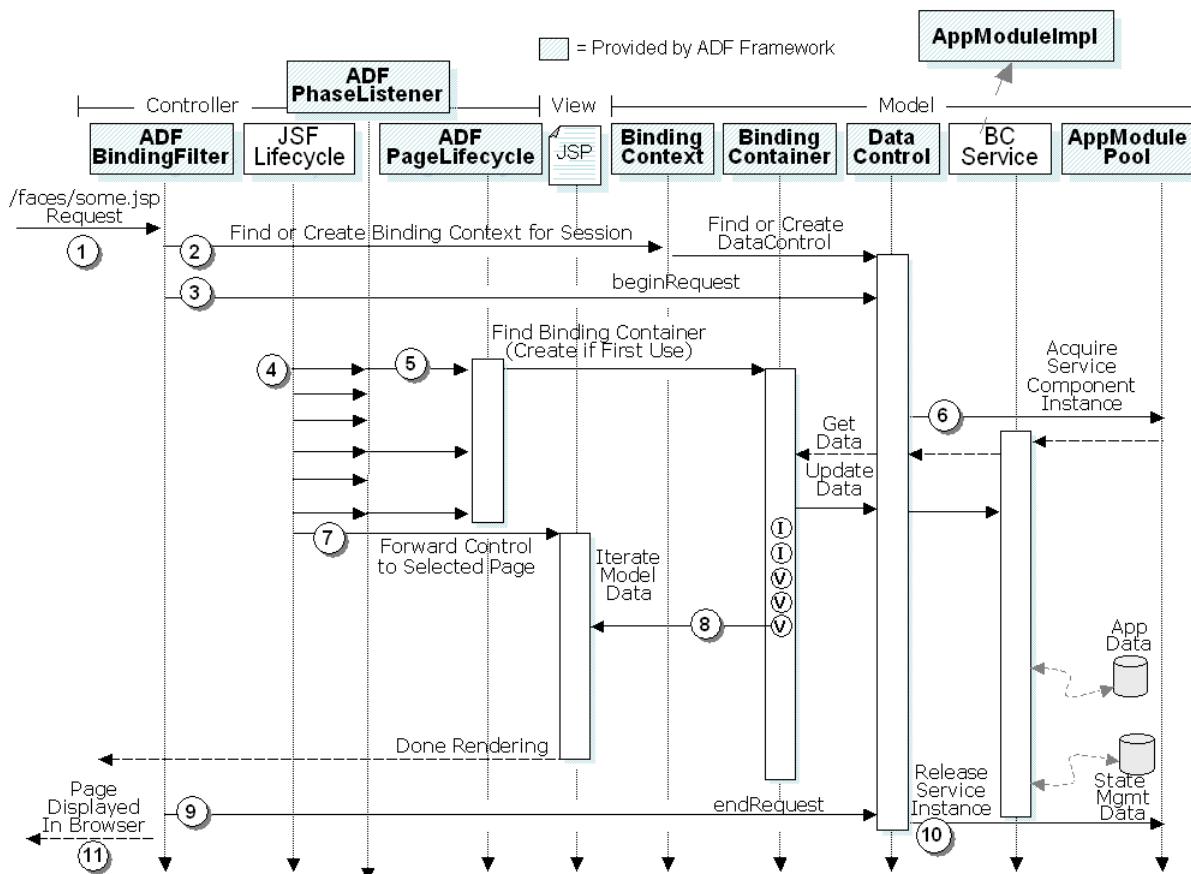
19.1 Introduction to the Fusion Page Lifecycle

When a page is submitted and a new page requested, the application invokes both the ADF Faces page lifecycle, which extends the standard JSF request lifecycle, and the ADF page lifecycle. The extended JSF lifecycle handles submitting the values on the page, validating component values, navigating pages, displaying components on the resulting page, and saving and restoring state. The JSF lifecycle phases use a UI component tree to manage the display of the faces components. This tree is a runtime representation of a JSF page: each UI component tag in a page corresponds to a UI component instance in the tree. The `FacesServlet` servlet manages the request processing lifecycle in JSF applications. `FacesServlet` creates an object called `FacesContext`, which contains the information necessary for request processing, and invokes an object that executes the lifecycle. For more details about the extended JSF lifecycle, see the "Understanding the JSF and ADF Faces Lifecycles" chapter of the *Oracle Fusion Middleware Web User Interface Developer's Guide for Oracle Application Development Framework*.

The ADF page lifecycle handles preparing and updating the data model, validating the data at the model layer, and executing methods on the business layer. The ADF page lifecycle uses the binding container to make data available for easy referencing by the page during the current page request.

The combined JSF and ADF page lifecycle is only one sequence within a larger sequence of events that begins when an HTTP request arrives at the application server and continues until the page is returned to the client. This overall sequence of events can be called the web page lifecycle. It follows processing through the model, view, and controller layers as defined by the MVC architecture. The page lifecycle is not a rigidly defined set of events, but is rather a set of events for a typical use case.

[Figure 19-1](#) shows a sequence diagram of the lifecycle of a web page request using JSF and Oracle ADF in tandem.

Figure 19–1 Lifecycle of a Web Page Request Using JSF and Oracle ADF

The basic flow of processing a web page request using JSF and Oracle ADF happens as follows:

1. A web request for `http://yourserver/yourapp/faces/some.jsp` arrives from the client to the application server.
2. The `ADFBindingFilter` object looks for the ADF binding context in the HTTP session, and if it is not yet present, initializes it for the first time. Some of the functions of the `ADFBindingFilter` include finding the name of the binding context metadata file, and finding and constructing an instance of each data control.
3. The `ADFBindingFilter` object invokes the `beginRequest()` method on each data control participating in the request. This method gives the data control a notification at the start of every request so that it can perform any necessary setup.
4. The JSF Lifecycle object, which is responsible for orchestrating the standard processing phases of each request, notifies the `ADFPhaseListener` class during each phase of the lifecycle, so that it can perform custom processing to coordinate the JSF lifecycle with the ADF Model data binding layer. For more information about the details of the JSF and ADF page lifecycle phases, see [Section 19.2, "The JSF and ADF Page Lifecycles"](#).

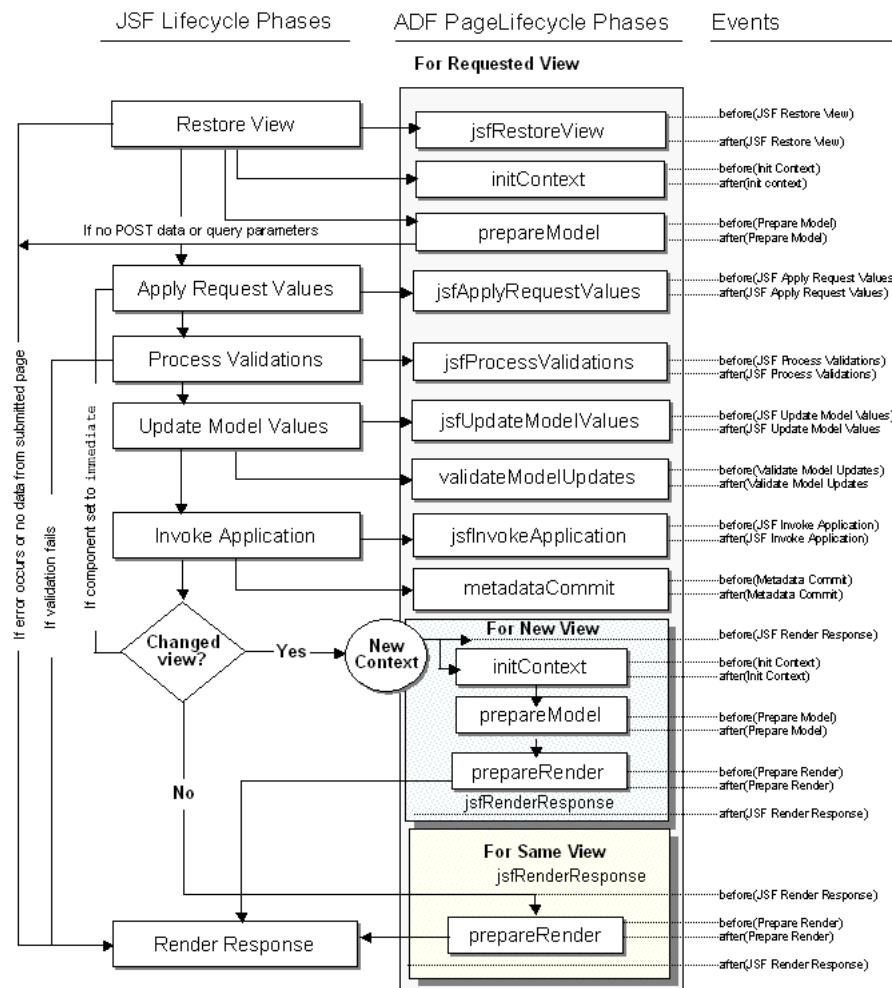
Note: The FacesServlet class (in `javax.faces.webapp`), configured in the `web.xml` file of a JSF application, is responsible for initially creating the JSF Lifecycle class (in `javax.faces.lifecycle`) to handle each request. However, since it is the Lifecycle class that does all the interesting work, the FacesServlet class is not shown in the diagram.

5. The ADFPhaseListener object creates an ADF PageLifecycle object to handle each request and delegates appropriate before and after phase methods to corresponding methods in the ADF PageLifecycle class. If the appropriate binding container for the page has never been used before during the user's session, it is created.
6. The first time an application module data control is referenced during the request, it acquires an instance of the application module from the application module pool.
7. The JSF Lifecycle object forwards control to the page to be rendered.
8. The UI components on the page access value bindings and iterator bindings in the page's binding container and render the formatted output to appear in the browser.
9. The ADFBindingFilter object invokes the `endRequest()` method on each data control participating in the request. This method gives a data control notification at the end of every request, so that they can perform any necessary resource cleanup.
10. An application module data control uses the `endRequest` notification to release the instance of the application module back to the application module pool.
11. The user sees the resulting page in the browser.

The ADF page lifecycle also contains phases that are defined simply to notify ADF page lifecycle listeners before and after the corresponding JSF phase is executed (that is, there is no implementation for these phases). These phases allow you to create custom listeners and register them with any phase of both the JSF and ADF page lifecycles, so that you can customize the ADF page lifecycle if needed, both globally or at the page level.

19.2 The JSF and ADF Page Lifecycles

Figure 19-2 shows how the JSF and ADF phases integrate in the lifecycle of a page request. For more information about how the JSF lifecycle operates on its own, see the "Understanding the JSF and ADF Faces Lifecycles" chapter of the *Oracle Fusion Middleware Web User Interface Developer's Guide for Oracle Application Development Framework*.

Figure 19–2 Lifecycle of a Page Request in a Fusion Web Application

In a JSF application that uses the ADF Model layer, the phases in the page lifecycle are as follows:

- **Restore View:** The URL for the requested page is passed to the bindingContext object, which finds the page definition file that matches the URL. The component tree of the requested page is either newly built or restored. All the component tags, event handlers, converters, and validators on the submitted page have access to the FacesContext instance. If the component tree is empty, (that is, there is no data from the submitted page), the page lifecycle proceeds directly to the Render Response phase.

If any discrepancies between the request state and the server-side state are detected, an error will be thrown and the page lifecycle jumps to the Render Response phase.

- **JSF Restore View:** Provides before and after phase events for the Restore View phase. You can create a listener and register it with the before or after event of this phase, and the application will behave as if the listener were registered with the Restore View phase. The Initialize Context phase of the ADF Model page lifecycle listens for the after (JSF Restore View) event and then executes. The ADF Controller uses listeners for the before and after events of this phase to synchronize the server-side state with the request. For example, it is in this phase

that browser back button detection and bookmark reference are handled. After the before and after listeners are executed, the page flow scope is available.

- Initialize Context: The page definition file is used to create the `bindingContainer` object, which is the runtime representation of the page definition file for the requested page. The `LifecycleContext` class used to persist information throughout the ADF page lifecycle phases is instantiated and initialized with values for the associated request, binding container, and lifecycle.
- Prepare Model: The ADF page lifecycle enters the Prepare Model phase by calling the `BindingContainer.refresh(PREPARE_MODEL)` method. During the Prepare Model phase, BindingContainer page parameters are prepared and then evaluated. If parameters for a task flow exist, they are passed into the flow.

Next, any executables that have their `refresh` attribute set to `prepareModel` are refreshed based on the order of entry in the page definition file's `<executables>` section and on the evaluation of their `RefreshCondition` properties (if present). When an executable leads to an iterator binding refresh, the corresponding data control will be executed, and that leads to execution of one or more collections in the service objects. If an iterator binding fails to refresh, a JBO exception will be thrown and the data will not be available to display. For more information, see [Section 19.2.1, "What You May Need to Know About Using the Refresh Property Correctly"](#).

If the incoming request contains no POST data or query parameters, then the lifecycle forwards to the Render Response phase.

If the page was created using a template, and that template contains bindings using the ADF Model layer, the template's page definition file is used to create the binding container for the template. The container is then added to the binding context.

If any `taskFlow` executable bindings exist (for example, if the page contains a region), the `taskFlow` binding creates an ADF Controller `ViewPortContext` object for the task flow, and any nested binding containers for pages in the flow are then executed.

- Apply Request Values: Each component in the tree extracts new values from the request parameters (using its `decode` method) and stores those values locally. Most associated events are queued for later processing. If you have set a component's `immediate` attribute to `true`, then the validation, conversion, and events associated with the component are processed during this phase and the lifecycle skips the Process Validations, Update Model Values, and Invoke Application phases. Additionally, any associated iterators are invoked. For more information about ADF Faces validation and conversion, see the "Validating and Converting Input" chapter in the *Oracle Fusion Middleware Web User Interface Developer's Guide for Oracle Application Development Framework*.
- JSF Apply Request Values: Provides before and after phase events for the Apply Request Values phase. You can create a listener and register it with the before or after event of this phase, and the application will behave as if the listener were registered with the Apply Request Values phase.
- Process Validations: Local values of components are converted and validated on the client. If there are errors, the lifecycle jumps to the Render Response phase. At the end of this phase, new component values are set, any validation or conversion error messages and events are queued on `FacesContext`, and any value change events are delivered. Exceptions are also caught by the binding container and cached.

- **JSF Process Validations:** Provides before and after phase events for the Process Validations phase. You can create a listener and register it with the before or after event of this phase, and the application will behave as if the listener were registered with the Process Validations phase.
- **Update Model Values:** The component's validated local values are moved to the model and the local copies are discarded. For any updateable components (such as an `inputText` component), corresponding iterators are refreshed, if the refresh condition is set to the default (`deferred`) and the refresh condition (if any) evaluates to true.

If you are using a backing bean for a JSF page to manage your UI components, any UI attributes bound to a backing bean property will also be refreshed in this phase.
- **JSF Update Model Values:** Provides before and after phase events for the Update Model Values phase. You can create a listener and register it with the before or after event of this phase, and the application will behave as if the listener were registered with the Update Model Values phase.
- **Validate Model Updates:** The updated model is now validated against any validation routines set on the model. Exceptions are caught by the binding container and cached.
- **Invoke Application:** Any action bindings for command components or events are invoked.
- **JSF Invoke Application:** Provides before and after phase events for the Invoke Application phase. You can create a listener and register it with the before or after event of this phase, and the application will behave as if the listener were registered with the Invoke Application phase.
- **Metadata Commit:** Changes to runtime metadata are committed.
- **Initialize Context** (only if navigation occurred in the Invoke Application lifecycle): The Initialize Context phase listens for the `beforeJSFRenderResponse` event to execute. The page definition file for the next page is initialized.
- **Prepare Model** (only if navigation occurred in the Invoke Application lifecycle): Any page parameters contained in the next page's definition are set.
- **Prepare Render:** The binding container is refreshed to allow for any changes that may have occurred in the Apply Request Values or Validation phases. Any iterators that correspond to read-only components (such as an `outputText` component) are refreshed. The `prepareRender` event is sent to all registered listeners, as is the `afterJSFRenderResponse` event.

Note: Instead of displaying `prepareRender` as a valid phase for a selection, JDeveloper displays `renderModel`, which represents the `refresh (RENDER_MODEL)` method called on the binding container.

- **Render Response:** The components in the tree are rendered as the Java EE web container traverses the tags in the page. State information is saved for subsequent requests and the Restore View phase.

In order to lessen the wait time required to display both a page and any associated data, certain ADF Faces rich client components such as the `table` component, use data streaming for their initial request. When a page contains one or more of these components, the page goes through the normal lifecycle. However, instead of fetching the data during that request, a special separate request is run. Because the

page has just rendered, only the Render Response phase executes for the components that use data streaming, and the corresponding data is fetched and displayed. If the user's action (for example scrolling in a table), causes a subsequent data fetch another request is executed. Tables, trees, tree tables, and data visualization components all use data streaming.

- JSF Render Response: Provides before and after phase events for the Render Response phase. You can create a listener and register it with the before or after event of this phase, and the application will behave as if the listener were registered with the Render Response phase.

19.2.1 What You May Need to Know About Using the Refresh Property Correctly

The Refresh and RefreshCondition attributes are used to determine when and whether to invoke an executable. Refresh determines the phase in which to invoke the executable, while the refresh condition determines whether the condition has been met. By default, when JDeveloper adds an executable to a page definition (for example, when you drop an operation as a command component), the Refresh attribute for the executable binding is set to deferred, which enforces execution whenever demanded the first time, (for example, when a binding value is referenced in an EL expression). If no RefreshCondition value exists, the executable is invoked. If a value for RefreshCondition exists, then that value is evaluated, and if the return value of the evaluation is true, then the executable is invoked. If the value evaluates to false, the executable is not invoked. For details about the refresh attribute, see [Section A.7.1, "PageDef.xml Syntax"](#).

For most cases in a Fusion web application, you should not change the refresh condition on an iterator binding or a taskFlow executable binding. Additionally, instead of inserting invokeAction executables into a page definition, you should use method activities in a task flow to invoke methods before a page renders. However, there may be cases when you do need to add an invokeAction executable and set the refresh property correctly.

The valid values for the Refresh property of an invokeAction are as follows:

- prepareModel: Executes the invokeAction executable during the Prepare Model phase.
- renderModel: Executes the invokeAction executable during the Prepare Render phase.

Tip: Notice in [Figure 19–2](#) that the key distinction between the Prepare Model phase and the Prepare Render phase is that one comes before JSF's Invoke Application phase, and one after. Since JSF's Invoke Application phase is when action listeners fire, if you need the method or operation associated with the invokeAction to execute after these action listeners have performed their processing, you'll want to set the Refresh property to renderModel.

- ifNeeded: Executes the invokeAction if needed, based on the refreshCondition attribute or, if no value is supplied, during the Prepare Model phase. To determine if the execution is needed, it performs an optimization to compare the current set of evaluated parameter values with the set that was used to invoke the method action binding previously. If the parameter values for the current invocation are exactly the same as those used previously, the invokeAction does not invoke its bound method action binding. Use this setting if the invokeAction executable binds to a method action binding that accepts parameters.

Tip: For `invokeAction` executables that are bound to methods that do not take parameters, the `invokeAction` will be called twice. To use the `invokeAction` executable with parameterless methods, you should use the `RefreshCondition` attribute so that the condition evaluates to only invoke the method if the value has changed. This will prevent multiple invocations.

- `prepareModelIfNeeded` and `renderModelIfNeeded`: Same as `ifNeeded`, except that the `invokeAction` is executed during the named phase.

Tip: Any `invokeAction` executable in a page definition file must have a value other than the default (`deferred`) for its `refresh` attribute, or it will not be refreshed and invoked.

Other values of `Refresh` are either not relevant to `invokeAction` objects (such as `Never`), or reserved for future use. For information about the other settings, see [Table A-4 in Appendix A, "Oracle ADF XML Files"](#).

Tip: You can determine the order of executable invocation using the `refreshAfter` attribute. For example, say you have two `invokeAction` elements; one with an ID of `myAction` and another with an ID of `anotherAction`, and you want `myAction` to fire after `anotherAction`. You would set the `refreshAfter` condition on `myAction` to `anotherAction`.

19.2.2 What You May Need to Know About Task Flows and the Lifecycle

The task flows associated with ADF Regions are initially refreshed when their parent page is first displayed. At that time, the task flow's parameter values are passed in from the parent page to the region, and the initial page fragment within the region is displayed. The task flow's bindings will only be refreshed based on its `Refresh` and `RefreshCondition` attributes.

Tip: If you have a region on a page that is not initially disclosed (for example, a popup dialog), the parameters still need to be available when the parent page is rendered, even though the region might not be displayed. If a region requires parameters, but those parameter values will not be available when the parent page is rendered, then you should use a router activity as the initial activity in the region's flow. If the parameters are null, the flow should not continue and the region will not display. If the parameter values are available, the flow should forward to the first view activity.

If you set an EL expression as the value of the `RefreshCondition` attribute, it will be evaluated during the Prepare Render phase of the lifecycle. When the expression evaluates to `true`, the task flow will be refreshed again. When `RefreshCondition` evaluates to `false`, the behavior is the same as if the `RefreshCondition` had not been specified.

Note: If the variable `bindings` is used within the EL expression, the context refers to the binding container of the parent page, not the page fragment displayed within the region.

The valid values for the `Refresh` property of a task flow executable are as follows:

- `default`: The region will be refreshed only once, when the parent page is first displayed.
- `ifNeeded`: Refreshes the region only if there has been a change to `taskFlow` Binding parameter values. If the `taskFlow` binding does not have parameters, then `ifNeeded` is equivalent to the `default`. When `ifNeeded` is used, the `RefreshCondition` attribute is not considered.

Note: `ifNeeded` is not supported when you pass parameters to the `taskFlow` Binding using a dynamic parameter Map. Instead, use `RefreshCondition="#{EL.Expression}"`.

Because the only job of the `taskFlow` binding is to refresh its parameters, `Refresh="always"` doesn't apply. If the `taskFlow` binding's parameters don't change, there is no reason to refresh the ADF Region. The `Refresh` attribute does not need to be specified: the `RefreshCondition` and `Refresh` attributes are mutually exclusive. Note that setting the `Refresh` attribute to `ifNeeded` takes precedence over any value for the `RefreshCondition` attribute.

Note that the child page fragment's page definition still handles the refresh of the bindings of the child page fragments.

19.3 Object Scope Lifecycles

At runtime, ADF objects such as the binding container and managed beans are instantiated. Each of these objects has a defined lifespan set by its scope attribute. You can access a scope as a `java.util.Map` from the `RequestContext` API. For example, to access an object named `foo` in the request scope, you would use the expression `#{{requestScope.foo}}`.

There are six types of scopes in a Fusion web application:

- Application scope: The object is available for the duration of the application.
- Session scope: The object is available for the duration of the session.

Note: There's no window uniqueness for session scope, all windows in the session share the same session scope instance. If you are concerned about multiple windows being able to access the same object (for example to ensure that managed beans do not conflict across windows), you should use a scope that is window-specific, such as page flow or view scope.

- Page flow scope: The object is available for the duration of a bounded task flow.

Note: Because these are not standard JSF scopes, you cannot register a managed bean to these scopes. However, the value of a managed bean property can refer to values in these scopes.

Also, EL expressions must explicitly include the scope to retrieve values. For example, to retrieve `foo` from the `pageFlowScope` scope, your expression would be `#{{pageFlowScope.foo}}`.

- Request scope: The object is available from the time an HTTP request is made until a response is sent back to the client.
- Backing bean scope: Used for managed beans for page fragments and declarative components only, the object is available from the time an HTTP request is made until a response is sent back to the client. This scope is needed for fragments and declarative components because there may be more than one page fragment or declarative component on a page, and to prevent collisions, any values must be kept in separate scope instances. Therefore, any managed bean for a page fragment or declarative component must use backing bean scope.

Note: Because these are not standard JSF scopes, you cannot register a managed bean to these scopes. However, the value of a managed bean property can refer to values in these scopes.

Also, EL expressions must explicitly include the scope to retrieve values. For example, to retrieve `foo` from the backing bean scope, your expression would be `#{BackingBeanScope.foo}`.

- View scope: The object is available until the view ID for the current view activity changes. You can use view scope to hold values for a given page. While request scope can be used to store a value needed from one page to the next, anything stored in view scope will be lost once the view ID changes.

Note: Because these are not standard JSF scopes, you cannot register a managed bean to these scopes. However, the value of a managed bean property can refer to values in these scopes.

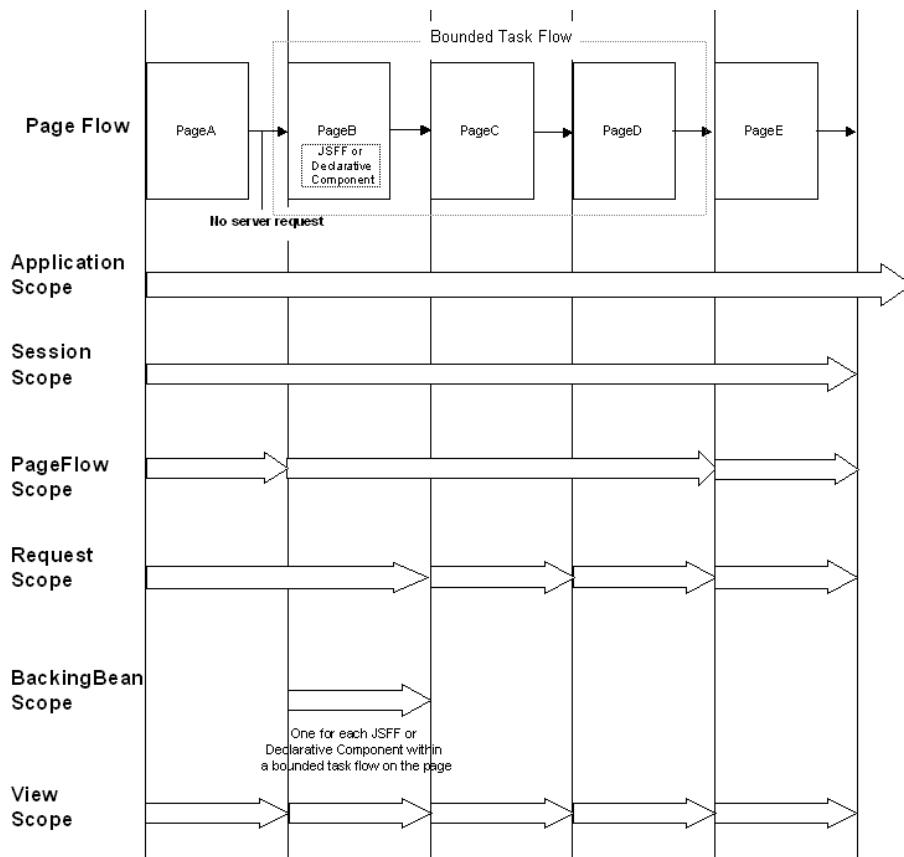
Also, EL expressions must explicitly include the scope to retrieve values. For example, to retrieve `foo` from the view scope, your expression would be `#{viewScope.foo}`.

Note: When you create objects (such as a managed bean) that require you to define a scope, you can set the scope to `none`, meaning that it will not live within any particular scope, but will instead be instantiated each time it is referenced.

Object scopes are analogous to global and local variable scopes in programming languages. The wider the scope, the higher availability of an object. During their life, these objects may expose certain interfaces, hold information, or pass variables and parameters to other objects. For example, a managed bean defined in session scope will be available for use during multiple page requests. However, a managed bean defined in request scope will only be available for the duration of one page request.

By default, the binding container and the binding objects it contains are defined in session scope. However, the *values* referenced by value bindings and iterator bindings are undefined between requests and for scalability reasons do not remain in session scope. Therefore, the values that binding objects refer to are valid only during a request in which that binding container has been prepared by the ADF lifecycle. What stays in session scope are only the binding container and binding objects themselves.

Figure 19–3 shows the time period during which each type of scope is valid.

Figure 19–3 Relationship Between Scopes and Page Flow

When determining what scope to register a managed bean with, always try to use the narrowest scope possible. Only use the session scope for information that is relevant to the whole session, such as user or context information. Avoid using session scope to pass values from one task flow to another. When creating a managed bean for a page fragment or a declarative component, you must use backing bean scope.

19.3.1 What You May Need to Know About Object Scopes and Task Flows

When determining what scope to use for variables within a task flow, you should use any of the scope options other than application or session scope. These two scopes will persist objects in memory beyond the life of the task flow and therefore compromise the encapsulation and reusable aspects of a task flow. In addition, application and session scopes may keep objects in memory longer than needed, causing unneeded overhead.

When you need to pass data values between activities within a task flow, you should use page flow scope. View scope is recommended for variables that are needed only within the current view activity, not across view activities. Request scope should be used when the scope does not need to persist longer than the current request. It is the only scope that should be used to store UI component information. Lastly, backing bean scope must be used for backing beans in your task flow if there is a possibility that your task flow will appear in two region components or declarative components on the same page and you would like to achieve region instance isolations.

19.4 Customizing the ADF Page Lifecycle

The ADF lifecycle contains clearly defined phases that notify ADF lifecycle listeners before and after the corresponding JSF phase is executed. You can customize this lifecycle by creating a custom phase listener that invokes your needed code, and then registering it with the lifecycle to execute during one of these phases.

For example, you can create a custom listener that executes custom code and then register it with the JSFApplyRequestValues phase so that it can be invoked either before or after the Apply Request Values phase.

Note: An application cannot have multiple phase listener instances. An initial `ADFPhaseListener` instance is by default, registered in the `META-INF/faces-config.xml` configuration file. Registering, for example, a customized subclass of the `ADFPhaseListener` creates a second instance. In this scenario, only the instance that was most recently registered is used.

The following warning message indicates when an instance has been replaced by a newer one: "ADFc: Replacing the ADF Page Lifecycle implementation with *class name of the new listener*."

19.4.1 How to Create a Custom Phase Listener

To create a custom phase listener, you must create a listener class that implements the `PagePhaseListener` interface. You then add methods that execute code either before or after the phase that the code needs to execute.

[Example 19–1](#) contains a template that you can modify to create a custom phase listener. See [Section 4.12.1, "How to Generate Custom Classes"](#) for more information about creating a class in JDeveloper.

Example 19–1 Example Custom Phase Listener

```
public class MyPagePhaseListener implements PagePhaseListener
{
    public void afterPhase(PagePhaseEvent event)
    {
        System.out.println("In afterPhase " + event.getPhaseId());
    }

    public void beforePhase(PagePhaseEvent event)
    {
        System.out.println("In beforePhase " + event.getPhaseId());
    }
}
```

Once you create the custom listener class, you need to register it with the phase in which the class needs to be invoked. You can either register it globally (so that the whole application can use it), or you can register it only for a single page.

19.4.2 How to Register a Listener Globally

To customize the ADF lifecycle globally, register your custom phase listener by editing the `adf-settings.xml` configuration file. The `adf-settings.xml` file is shared by several ADF components, including ADF Controller, to store configuration information. If the `adf-settings.xml` file does not yet exist, you need to create it.

For information, see the "ADF Faces Configuration" appendix of the *Oracle Fusion Middleware Web User Interface Developer's Guide for Oracle Application Development Framework*.

To register the listener in adf-settings.xml:

1. In the adf-settings.xml file, scroll down to

```
<adfc-controller-config xmlns=
    "http://xmlns.oracle.com/adf/controller/config">
```

If this entry does not exist, add it to the file as shown in [Example 19–2](#).

Example 19–2 adf-settings.xml Configuration File with Listener Registration

```
<?xml version="1.0" encoding="US-ASCII" ?>
<adf-config xmlns="http://xmlns.oracle.com/adf/config">
    .
    .
    .
    <adfc-controller-config xmlns="http://xmlns.oracle.com/adf/controller/config">
        <lifecycle>
            <phase-listener>
                <listener-id>MyPagePhaseListener</listener-id>
                <class>mypackage.MyPagePhaseListener</class>
            </phase-listener>
        </lifecycle>
    </adfc-controller-config>
    .
    .
    .
</adf-config>
```

2. Enter the remaining elements shown in italics in [Example 19–2](#).

3. Add values for the following elements:

- <listener-id> A unique identifier for the listener (you can use the fully qualified class name)
- <class> The class name of the listener

19.4.3 What You May Need to Know About Listener Order

You can specify multiple phase listeners in the adf-settings.xml and, optionally, the relative order in which they are called. When registering a new listener in the file, the position in the list of listeners is determined using two parameters:

- beforeIdSet: The listener is called before any of the listeners specified in beforeIdSet
- afterIdSet: The listener is called after any of the listeners specified in afterIdSet

[Example 19–3](#) contains an example configuration file in which multiple listeners have been registered for an application.

Example 19–3 adf-settings.xml Configuration File with Multiple Listener Registration

```
<lifecycle>
    <phase-listener>
        <listener-id>MyPhaseListener</listener-id>
        <class>view.myPhaseListener</class>
```

```

<after-id-set>
    <listener-id>ListenerA</listener-id>
    <listener-id>ListenerC</listener-id>
</after-id-set>
<before-id-set>
    <listener-id>ListenerB</listener-id>
    <listener-id>ListenerM</listener-id>
    <listener-id>ListenerY</listener-id>
</before-id-set>
</phase-listener>
</lifecycle>

```

In the example, MyPhaseListener is a registered listener that executes after listeners A and C but before listeners B, M, and Y. To execute MyPhaseListener after listener B, move the `<listener-id>` element for listener B under the `<after-id-set>` element.

19.4.4 How To Register a Lifecycle Listener for a Single Page

To customize the lifecycle of a single page, you set the `ControllerClass` attribute on the page definition file. This listener will be valid only for the lifecycle of the particular page described by the page definition. For more information about the page definition file and its role in a Fusion web application, see [Section 11.6, "Working with Page Definition Files"](#).

You specify a different controller class depending on whether it is for a standard JSF page or a page fragment:

To customize the ADF Lifecycle for a single page or page fragment:

1. In the Application Navigator, right-click the page or page fragment and choose **Go To Page Definition**.
2. In the Structure window, select the page definition node.
3. In the Property Inspector, click the dropdown menu next to the `ControllerClass` field and choose **Edit**.
4. Click **Hierarchy** and navigate to the appropriate controller class for the page or page fragment. Following are the controller classes to use for different types of pages:
 - Standard JSF page - specify
`oracle.adf.controller.v2.lifecycle.PageController`
 If you need to receive `afterPhase/beforePhase` events, specify
`oracle.adf.controller.v2.lifecycle.PagePhaseListener`
 - Page fragment - specify
`oracle.adf.model.RegionController`

Tip: You can specify the value of the page definition's `ControllerClass` attribute as a fully qualified class name using the method above, or you can enter an EL expression that resolves to a class directly in the `ControllerClass` field.

When using an EL expression for the value of the `ControllerClass` attribute, the Structure window may show a warning indicating that `"#{YourExpression}"` is not a valid class. You can safely ignore this warning.

19.4.5 What You May Need to Know About Extending RegionController for Page Fragments

Bindings inside page fragments used as regions are refreshed through the `refreshRegion` and `validateRegion` events of the `RegionController` interface. These events are available if you specify `oracle.adf.model.RegionController` in the `ControllerClass` field as described in [Section 19.4.4](#) above.

As shown in [Example 19–4](#), you can use the `refreshRegion` event to add custom code that executes before the region is refreshed. For example, you may want to refresh the bindings used by the page fragment in the region so that the refreshed binding values are propagated to the inner binding container.

To do this, create a new class that implements the `RegionController` interface. Then, write the following `refreshRegion` method, including your custom code that you want to execute before the Prepare Model phase.

Example 19–4 regionRefresh Method

```
public boolean refreshRegion(RegionContext regionCtx)
{
    int refreshFlag = regionCtx.getRefreshFlag();
    if (refreshFlag == RegionBinding.PREPARE_MODEL)
    {
        // Execute some code before
    }
    // Propagate the refresh to the inner binding container
    regionCtx.getRegionBinding().refresh(refreshFlag);

    return false;
}

public boolean validateRegion(RegionContext regionCtx)
{
    // Propagate the validate to the inner binding container
    regionCtx.getRegionBinding().validate();

    return false;
}
```

As shown in [Example 19–4](#), the refresh flag value can be:

- `RegionBinding.PREPARE_MODEL` - corresponds to the event occurring during the ADF Lifecycle `PREPARE_MODEL` phase
- `RegionBinding.RENDER_MODEL` - corresponds to the event occurring during the ADF Lifecycle `PREPARE_RENDER` phase

20

Creating a Basic Databound Page

This chapter describes how to use the Data Controls panel to create databound forms using ADF Faces components.

This chapter includes the following sections:

- [Section 20.1, "Introduction to Creating a Basic Databound Page"](#)
- [Section 20.2, "Using Attributes to Create Text Fields"](#)
- [Section 20.3, "Creating a Basic Form"](#)
- [Section 20.4, "Incorporating Range Navigation into Forms"](#)
- [Section 20.5, "Creating a Form to Edit an Existing Record"](#)
- [Section 20.6, "Creating an Input Form"](#)
- [Section 20.7, "Using a Dynamic Form to Determine Data to Display at Runtime"](#)
- [Section 20.8, "Modifying the UI Components and Bindings on a Form"](#)

20.1 Introduction to Creating a Basic Databound Page

You can create UI pages that allow you to display and collect information using data controls created for your business services. For example, using the Data Controls panel, you can drag an attribute for an item, and then choose to display the value either as read-only text or as an input text field with a label. JDeveloper creates all the necessary JSF tag and binding code needed to display and update the associated data. For more information about the Data Controls panel and the declarative binding experience, see [Chapter 11, "Using Oracle ADF Model in a Fusion Web Application"](#).

Instead of having to drop individual attributes, JDeveloper allows you to drop all attributes for an object at once as a form. The actual UI components that make up the form depend on the type of form dropped. You can create forms that display values, forms that allow users to edit values, and forms that collect values (input forms).

For example, the StoreFront module contains a page that allows users to register information about themselves. This form was created by dragging and dropping the `CustomerRegistration` collection from the Data Controls panel.

Once you drop the UI components, you can then drop built-in operations as command UI components that allow you to navigate through the records in a collection or that allow users to operate on the data, such as committing, deleting, or creating a record. For example, you can create a button that allows users to delete data objects displayed in the form. You can also modify the default components to suit your needs.

20.2 Using Attributes to Create Text Fields

JDeveloper allows you to create text fields declaratively in a WYSIWYG development environment for your JSF pages, meaning you can design most aspects of your pages without needing to look at the code. When you drag and drop items from the Data Controls panel, JDeveloper declaratively binds ADF Faces text UI components to attributes on a data control using an attribute binding.

20.2.1 How to Create a Text Field

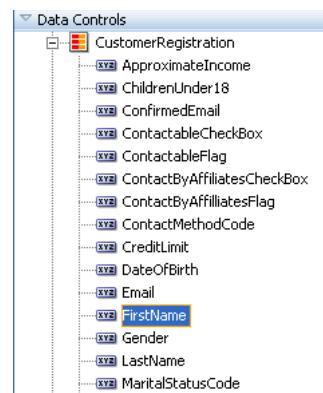
To create a text field that can display or update an attribute, you drag and drop an attribute of a collection from the Data Controls panel.

To create a bound text field:

- From the Data Controls panel, select an attribute for a collection. For a description of the icons that represent attributes and other objects in the Data Controls panel, see [Section 11.3, "Using the Data Controls Panel"](#).

For example, [Figure 20–1](#) shows the FirstName attribute under the CustomerRegistration collection of the StoreServiceAMDataControl data control in the StoreFront module. This is the attribute to drop to display or enter the customer's first name.

Figure 20–1 Attributes Associated with a Collection in the Data Controls Panel



- Drag the attribute onto the page, and from the context menu choose the type of widget to display or collect the attribute value. For an attribute, you are given the following choices:

- **Texts:**

- **ADF Input Text w/ Label:** Creates an ADF Faces `inputText` component with a nested `validator` component. The `label` attribute is populated.

Tip: For more information about validators and other attributes of the `inputText` component, see the "Using Input Components and Defining Forms" chapter of the *Oracle Fusion Middleware Web User Interface Developer's Guide for Oracle Application Development Framework*.

- **ADF Input Text:** Creates an ADF Faces `inputText` component with a nested `validator` component. The `label` attribute is not populated.

- **ADF Output Text w/ Label:** Creates a panelLabelAndMessage component that holds an ADF Faces outputText component. The label attribute on the panelLabelAndMessage component is populated.
 - **ADF Output Text:** Creates an ADF Faces outputText component. No label is created.
 - **ADF Output Formatted w/Label:** Same as ADF Output Text w/Label, but uses an outputFormatted component instead of an outputText component. The outputFormatted component allows you to add a limited amount of HTML formatting. For more information, see the "Formatted Output Text" section of the *Oracle Fusion Middleware Web User Interface Developer's Guide for Oracle Application Development Framework*
 - **ADF Output Formatted:** Same as ADF Output Formatted w/Label, but without the label.
 - **ADF Label:** An ADF Faces outputLabel component.
- **List of Values:** Creates ADF LOV lists. For more information about how these lists work, see [Section 5.11, "Working with List of Values \(LOV\) in View Object Attributes"](#). For more information about using the lists on a JSF page, see [Section 23.1, "Creating a Selection List"](#).
 - **Single selections:** Creates single selection lists. For more information about creating lists on a JSF page, see [Section 23.1, "Creating a Selection List"](#).

Note: These selections are your choices by default. However, the list of components available to use for an attribute can be configured as a control hint on the associated entity or view object. For more information, see [Section 4.6, "Defining Attribute Control Hints for Entity Objects"](#) and [Section 5.12, "Defining Attribute Control Hints for View Objects"](#).

For the purposes of this chapter, only the text components (and not the lists) will be discussed.

20.2.2 What Happens When You Create a Text Field

When you drag an attribute onto a JSF page and drop it as a UI component, among other things, a page definition file is created for the page (if one does not already exist). For a complete account of what happens when you drag an attribute onto a page, see [Section 11.3.2, "What Happens When You Use the Data Controls Panel"](#). Bindings for the iterator and attributes are created and added to the page definition file. Additionally, the necessary JSPX page code for the UI component is added to the JSF page.

20.2.2.1 Creating and Using Iterator Bindings

Whenever you create UI components on a page by dropping an item that is part of a collection from the Data Controls panel (or you drop the whole collection as a form or table), JDeveloper creates an iterator binding if it does not already exist. 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 in order 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. In the case of ADF Business Components, the actual iterator is the default row set iterator for the default row set of the view object instance in the application module's data model.

For example, if you drop the FirstName attribute under the CustomerRegistration collection, JDeveloper creates an iterator binding for the CustomerRegistration collection.

Tip: There is one iterator binding created for each collection. This means that when you drop two attributes from the same collection (or drop the collection twice), they use the same binding. This is fine, unless you need the binding to behave differently for the different components. In that case, you will need to manually create separate iterator bindings.

The iterator binding's rangeSize attribute determines how many rows of data are fetched from a data control each time the iterator binding is accessed. This attribute gives you a relative set of 1-n rows positioned at some absolute starting location in the overall row set. By default, it is the attribute set to 25. For more information about using this attribute, see [Section 20.4.2.2, "Iterator RangeSize Attribute"](#). [Example 20–1](#) shows the iterator binding created when you drop an attribute from the CustomerRegistration collection.

Example 20–1 Page Definition Code for an Iterator Binding for an Attribute Dropped from a Collection

```
<executables>
    <iterator Binds="CustomerRegistration" RangeSize="25"
        DataControl="StoreServiceAMDataControl"
        id="CustomerRegistrationIterator"/>
</executables>
```

For information regarding the iterator binding element attributes, see [Appendix B, "Oracle ADF Binding Properties"](#).

This metadata allows the ADF binding container to access the attribute values. Because the iterator binding is an executable, by default, it is invoked when the page is loaded, thereby allowing the iterator to access and iterate over the CustomerRegistration collection. This means that the iterator will manage all the CustomerRegistration objects in the collection, including determining the current CustomerRegistration or range of CustomerRegistration objects.

20.2.2.2 Creating and Using Value Bindings

When you drop an attribute from the Data Controls panel, JDeveloper creates an attribute binding that is used to bind the UI component to the attribute's value. This type of binding presents the value of an attribute for a single object in the current row in the collection. Value bindings can be used both to display and to collect attribute values.

For example, if you drop the PrincipalName attribute under the CustomerRegistration collection as an ADF Output Text w/Label widget onto a page, JDeveloper creates an attribute binding for the PrincipalName attribute. This allows the binding to access the attribute value of the current record. [Example 20–2](#) shows the attribute binding for PrincipalName created when you drop the attribute from the CustomerRegistration collection. Note that the attribute value references the iterator named CustomerRegistrationIterator.

Example 20–2 Page Definition Code for an Attribute Binding

```
<bindings>
    ...
    <attributeValues IterBinding="CustomerRegistrationIterator"
        id="PrincipalName">
        <AttrNames>
            <Item Value="PrincipalName" />
        </AttrNames>
    </attributeValues>
</bindings>
```

For information regarding the attribute binding element properties, see [Appendix B, "Oracle ADF Binding Properties"](#).

20.2.2.3 Using EL Expressions to Bind UI Components

When you create a text field by dropping an attribute from the Data Controls panel, JDeveloper creates the UI component associated with the widget dropped by writing the corresponding tag to the JSF page.

For example, when you drop the PrincipalName attribute as an Output Text w/Label widget, JDeveloper inserts the tags for a panelLabelAndMessage component and an outputText component. It creates an EL expression that binds the label attribute of the panelLabelAndMessage component to the label property of hints created for the PrincipalName's binding. This expression evaluates to the label hint set on the view object (for more information about hints, see [Section 5.12, "Defining Attribute Control Hints for View Objects"](#)). It creates another expression that binds the outputText component's value attribute to the inputValue property of the PrincipalName binding, which evaluates to the value of the PrincipalName attribute for the current row.

[Example 20–3](#) shows the code generated on the JSF page when you drop the PrincipalName attribute as an Output Text w/Label widget.

Example 20–3 JSF Page Code for an Attribute Dropped as an Output Text w/Label

```
<af:panelLabelAndMessage label="#{bindings.PrincipalName.hints.label}">
    <af:outputText value="#{bindings.PrincipalName.inputValue}" />
</af:panelLabelAndMessage>
```

If instead you drop the PrincipalName attribute as an Input Text w/Label widget, JDeveloper creates an inputText component. As [Example 20–4](#) shows similar to the output text component, the value is bound to the inputValue property of the PrincipalName binding. Additionally, the following properties are also set:

- **label**: Bound to the label property of the control hint set on the object.
- **required**: Bound to the mandatory property of the control hint.
- **columns**: Bound to the displayWidth property of the control hint, which determines how wide the text box will be.
- **maxLength**: Bound to the precision property of the control hint. This control hint property determines the maximum number of characters per line that can be entered into the field.

In addition, JDeveloper adds a validator component.

Example 20–4 JSF Page Code for an Attribute Dropped as an Input Text w/Label

```
<af:inputText value="#{bindings.PrincipalName.inputValue}" />
```

```

        label="#{bindings.PrincipalName.hints.label}"
        required="#{bindings.PrincipalName.hints.mandatory}"
        columns="#{bindings.PrincipalName.hints.displayWidth}"
        maximumLength="#{bindings.PrincipalName.hints.precision}"
    <f:validator binding="#{bindings.PrincipalName.validator}" />
</af:inputText>
```

You can change any of these values to suit your needs. For example, the mandatory control hint on the view object is set to `false` by default, which means that the required attribute on the component will evaluate to `false` as well. You can override this value by setting the required attribute on the component to `true`. If you decide that all instances of the attribute should be mandatory, then you can change the control hint on the view object, and all instances will then be required. For more information about these properties, see [Appendix B, "Oracle ADF Binding Properties"](#).

20.3 Creating a Basic Form

Instead of dropping each of the individual attributes of a collection to create a form, you can a complete form that displays or collects data for all the attributes on an object.

For example, you could create a page that displays basic information about registered users in the StoreFront module by dragging and dropping the `CustomerInfoVO` collection.

You can also create forms that provide more functionality than simply displaying data from a collection. For information about creating a form that allows a user to update data, see [Section 20.5, "Creating a Form to Edit an Existing Record"](#). For information about creating forms that allow users to create a new object for the collection, see [Section 20.6, "Creating an Input Form"](#). You can also create search forms. For more information, see [Chapter 25, "Creating ADF Databound Search Forms"](#).

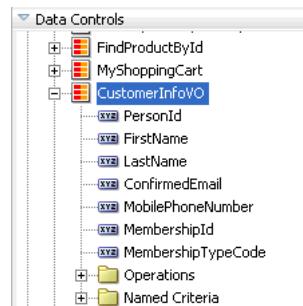
20.3.1 How to Create a Form

To create a form using a data control, you bind the UI components to the attributes on the corresponding object in the data control. JDeveloper allows you to do this declaratively by dragging and dropping a collection or a structured attribute from the Data Controls panel.

To create a basic form:

- From the Data Controls panel, select the collection that represents the data you wish to display. [Figure 20–2](#) shows the `CustomerInfoVO` collection for the `StoreServiceAMDataControl` data control.

Figure 20–2 CustomerInfo View Object in the Data Controls Panel



2. Drag the collection onto the page, and from the context menu choose the type of form that will be used to display or collect data for the object. For a form, you are given the following choices:
 - **ADF Form:** Launches the Edit Form Fields dialog that allows you to select individual attributes instead of having JDeveloper create a field for every attribute by default. It also allows you to select the label and UI component used for each attribute. By default, ADF `inputText` components are used for most attributes. Each `inputText` component has the `label` attribute populated.
 Attributes that are dates use the `InputDate` component. Additionally, if a control type control hint has been created for an attribute, or if the attribute has been configured to be a list, then the component set by the hint is used instead. `InputText` components contain a validator tag that allows you to set up validation for the attribute, and if the attribute is a number or a date, a converter is also included.

Tip: For more information about validators, converters, and other attributes of the `inputText` component, see the "Using Input Components and Defining Forms" chapter of the *Oracle Fusion Middleware Web User Interface Developer's Guide for Oracle Application Development Framework*.

You can elect to include navigational controls that allow users to navigate through all the data objects in the collection. For more information, see [Section 20.4, "Incorporating Range Navigation into Forms"](#). You can also include a **Submit** button used to submit the form. This button submits the HTML form and applies the data in the form to the bindings as part of the JSF/ADF page lifecycle. For additional help in using the dialog, click **Help**. All UI components are placed inside a `panelFormLayout` component.

 - **ADF Read-Only Form...**: Same as the ADF Form, but only read-only `outputText` components are used. Since the form is meant to display data, no validator tags are added (converters are included). Attributes of type `Date` use the `outputText` component when in a read-only form. All components are placed inside `panelLabelAndMessage` components, which have the `label` attribute populated, and are placed inside a `panelFormLayout` component.
 - **ADF Search Form:** Creates a form that can be used to execute a Query-by-Example (QBE) search. For more information, see [Chapter 25, "Creating ADF Databound Search Forms"](#).
 - **ADF Creation Form:** Creates an input form that allows users to create a new instance for the collection. For more information, see [Section 20.6, "Creating an Input Form"](#).
3. If you are building a form that allows users to update data, you now need to drag and drop an operation that will perform the update. For more information, see [Section 20.5, "Creating a Form to Edit an Existing Record"](#).

20.3.2 What Happens When You Create a Form

Dropping an object as a form from the Data Controls panel has the same effect as dropping a single attribute, except that multiple attribute bindings and associated UI components are created. The attributes on the UI components (such as `value`) are bound to properties on that attribute's binding object (such as `inputValue`) or to the

values of control hints set on the corresponding business object. [Example 20–5](#) shows some of the code generated on the JSF page when you drop the CustomerInfoVO collection as a default ADF Form.

Note: If an attribute is marked as hidden on the associated view or entity object, then no corresponding UI is created for it.

Example 20–5 Code on a JSF Page for an Input Form

```
<af:panelFormLayout>
    <af:inputText value="#{bindings.PersonId.inputValue}"
        label="#{bindings.PersonId.hints.label}"
        required="#{bindings.PersonId.hints.mandatory}"
        columns="#{bindings.PersonId.hints.displayWidth}"
        maximumLength="#{bindings.PersonId.hints.precision}">
        <f:validator binding="#{bindings.PersonId.validator}" />
        <af:convertNumber groupingUsed="false"
            pattern="#{bindings.PersonId.format}" />
    </af:inputText>
    <af:inputText value="#{bindings.FirstName.inputValue}"
        label="#{bindings.FirstName.hints.label}"
        required="#{bindings.FirstName.hints.mandatory}"
        columns="#{bindings.FirstName.hints.displayWidth}"
        maximumLength="#{bindings.FirstName.hints.precision}">
        <f:validator binding="#{bindings.FirstName.validator}" />
    </af:inputText>
    <af:inputText value="#{bindings.LastName.inputValue}"
        label="#{bindings.LastName.hints.label}"
        required="#{bindings.LastName.hints.mandatory}"
        columns="#{bindings.LastName.hints.displayWidth}"
        maximumLength="#{bindings.LastName.hints.precision}">
        <f:validator binding="#{bindings.LastName.validator}" />
    </af:inputText>
    .
    .
    .
</af:panelFormLayout>
```

Note: For information regarding the validator and converter tags, see the "Validating and Converting Input" chapter of the *Oracle Fusion Middleware Web User Interface Developer's Guide for Oracle Application Development Framework*.

When you choose to create an input form using an object that contains a defined list of values (LOV), then a `selectOneChoice` component is created instead of an `inputText` component. For example, the `PersonsVO` view object contains defined LOVs for the `Title`, `Gender`, `MaritalStatusCode`, and `PersonTypeCode` attributes. When you drop the `Persons` data control object as an ADF Form, instead of as an empty input text field, a dropdown list showing all values is created. For more information about how these lists work, see [Section 5.11, "Working with List of Values \(LOV\) in View Object Attributes"](#). For more information about using the lists on a JSF page, see [Section 23.1, "Creating a Selection List"](#).

Note: If the object contains a structured attribute (an attribute that is neither a Java primitive type nor a collection), that attribute will not appear in the dialog, and it will not have a corresponding component in the form. You will need to create those fields manually.

20.4 Incorporating Range Navigation into Forms

When you create an ADF Form, if you elect to include navigational controls, JDeveloper includes ADF Faces command components bound to existing navigational logic on the data control. This built-in logic allows the user to navigate through all the data objects in the collection.

20.4.1 How to Insert Navigation Controls into a Form

By default, when you choose to include navigation when creating a form using the Data Controls panel, JDeveloper creates **First**, **Last**, **Previous**, and **Next** buttons that allow the user to navigate within the collection.

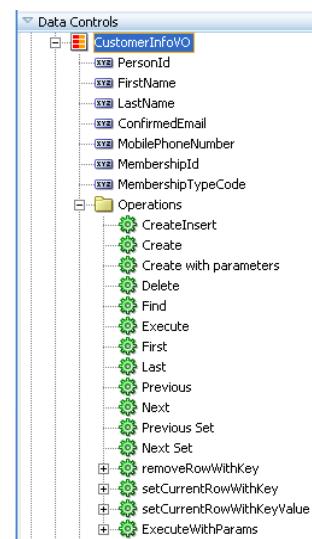
You can also add navigation buttons to an existing form manually.

To manually add navigation buttons:

- From the Data Controls panel, select the collection of objects on which you wish the operations to execute, and drag it onto the JSF page.

For example, if you want to navigate through a collection of persons, you would drag the **Next** operation associated with the Persons collection. [Figure 20–3](#) shows the operations associated with the `CustomerInfoVO` collection.

Figure 20–3 Operations Associated with a Collection



- From the ensuing context menu, choose either **ADF Button** or **ADF Link**.

Tip: You can also drop the **First**, **Previous**, **Next**, and **Last** buttons at once. To do so, drag the corresponding collection, and from the context menu, choose **Navigation > ADF Navigation Buttons**.

20.4.2 What Happens When You Create Command Buttons

When you drop any operation as a command component, JDeveloper:

- Defines an action binding in the page definition file for the associated operations
- Configures the iterator binding to use partial page rendering for the collection
- Inserts code in the JSF page for the command components

20.4.2.1 Using Action Bindings for Built-in Navigation Operations

Action bindings execute business logic. For example, they can invoke built-in methods on the action binding object. These built-in methods operate on the iterator or on the data control itself, and are represented as operations in the Data Controls panel.

JDeveloper provides navigation operations that allow users to navigate forward, backwards, to the last object in the collection, and to the first object.

Like value bindings, action bindings for operations contain a reference to the iterator binding when the action binding is bound to one of the iterator-level actions, such as Next or Previous. These types of actions are performed by the iterator, which determines the current object and can therefore determine the correct object to display when a navigation buttons is clicked. Action bindings to other than iterator-level actions, for example for a custom method on an application module, or for the commit or rollback operations, will not contain this reference.

Action bindings use the RequiresUpdateModel property, which determines whether or not the model needs to be updated before the action is executed. In the case of navigation operations, by default this property is set to true, which means that any changes made at the view layer must be moved to the model before navigation can occur. [Example 20–6](#) shows the action bindings for the navigation operations.

Example 20–6 Page Definition Code for an Operation Action Binding

```
<action IterBinding="CustomerInfoVOIterator" id="First"
        RequiresUpdateModel="true" Action="first"/>
<action IterBinding="CustomerInfoVOIterator" id="Previous"
        RequiresUpdateModel="true" Action="previous"/>
<action IterBinding="CustomerInfoVOIterator" id="Next"
        RequiresUpdateModel="true" Action="next"/>
<action IterBinding="CustomerInfoVOIterator" id="Last"
        RequiresUpdateModel="true" Action="last"/>
```

20.4.2.2 Iterator RangeSize Attribute

Iterator bindings have a rangeSize attribute that the binding uses to determine the number of data objects to make available for the page for each iteration. This attribute helps in situations when the number of objects in the data source is quite large. Instead of returning all objects, the iterator binding returns only a set number, which then become accessible to the other bindings. Once the iterator reaches the end of the range, it accesses the next set. [Example 20–7](#) shows the default range size for the CustomerInfoVO iterator.

Example 20–7 RangeSize Attribute for an Iterator

```
<iterator Binds="CustomerInfoVO" RangeSize="25"
          DataControl="StoreServiceAMDataControl"
          id="CustomerInfoVOIterator"
          ChangeEventPolicy="ppr" />
```

Note: This `rangeSize` attribute is not the same as the `rows` attribute on a table component. For more information, see [Table 21-1, "ADF Faces Table Attributes and Populated Values"](#).

By default, the `rangeSize` attribute is set to 25. This means that a user can view 25 objects, navigating back and forth between them, without needing to access the data source. The iterator keeps track of the current object. Once a user clicks a button that requires a new range (for example, clicking the **Next** button on object number 25), the binding object executes its associated method against the iterator, and the iterator retrieves another set of 25 records. The bindings then work with that set. You can change this setting as needed. You can set it to `-1` to have the full record set returned.

Note: When you create a navigateable form using the Data Controls panel, the `CacheResults` property on the associated iterator is set to `true`. This ensures that the iterator's state, including currency information, is cached between requests, allowing it to determine the current object. If this property is set to `false`, navigation will not work.

[Table 20-1](#) shows the built-in navigation operations provided on data controls and the result of invoking the operation or executing an event bound to the operation. For more information about action events, see [Section 20.4.3, "What Happens at Runtime: How Action Events and Action Listeners Work"](#).

Table 20-1 Built-in Navigation Operations

Operation	When invoked, the associated iterator binding will...
First	Move its current pointer to the beginning of the result set.
Last	Move its current pointer to the end of the result set.
Previous	Move its current pointer to the preceding object in the result set. If this object is outside the current range, the range is scrolled backward a number of objects equal to the range size.
Next	Move its current pointer to the next object in the result set. If this object is outside the current range, the range is scrolled forward a number of objects equal to the range size.
Previous Set	Move the range backward a number of objects equal to the range size attribute.
Next Set	Move the range forward a number of objects equal to the range size attribute.

20.4.2.3 Using EL Expressions to Bind to Navigation Operations

When you create command components using navigation operations, the command components are placed in a `panelGroupLayout` component. JDeveloper creates an EL expression that binds a navigational command button's `actionListener` attribute to the `execute` property of the action binding for the given operation.

At runtime an action binding will be an instance of the `FacesCtrlActionBinding` class, which extends the core `JUCtrlActionBinding` implementation class. The `FacesCtrlActionBinding` class adds the following methods:

- `public void execute(ActionEvent event)`: This is the method that is referenced in the `actionListener` property, for example `#{bindings.First.execute}`.

This expression causes the binding's operation to be invoked on the iterator when a user clicks the button. For example, the **First** command button's `actionListener` attribute is bound to the `execute` method on the **First** action binding.

- `public String outcome()`: This can be referenced in an `Action` property, for example `#{bindings.Next.outcome}`.

This can be used for the result of a method action binding (once converted to a `String`) as a JSF navigation outcome to determine the next page to navigate to.

Note: Using the `outcome` method on the action binding implies tying the view-controller layer too tightly to the model, so it should be rarely used.

Every action binding for an operation has an `enabled` boolean property that Oracle ADF sets to `false` when the operation should not be invoked. By default, JDeveloper binds the UI component's `disabled` attribute to this value to determine whether or not the component should be enabled. For example, the UI component for the **First** button has the following as the value for its `disabled` attribute:

```
# {!bindings.First.enabled}
```

This expression evaluates to `true` whenever the binding is not enabled, that is, when operation should not be invoked, thereby disabling the button. In this example, because the framework will set the `enabled` property on the binding to `false` whenever the first record is being shown, the **First** button will automatically be disabled because its `disabled` attribute is set to be `true` whenever `enabled` is `False`. For more information about the `enabled` property, see [Appendix B, "Oracle ADF Binding Properties"](#).

[Example 20–8](#) shows the code generated on the JSF page for navigation operation buttons. For more information about the `partialSubmit` attribute on the button, see [Section 20.4.2.4, "Using Automatic Partial Page Rendering"](#).

Example 20–8 JSF Code for Navigation Buttons Bound to ADF Operations

```
<af:facet name="footer">
  <af:panelGroupLayout>
    <af:commandButton actionListener="#{bindings.First.execute}"
      text="First"
      disabled="# {!bindings.First.enabled}"
      partialSubmit="true"/>
    <af:commandButton actionListener="#{bindings.Previous.execute}"
      text="Previous"
      disabled="# {!bindings.Previous.enabled}"
      partialSubmit="true"/>
    <af:commandButton actionListener="#{bindings.Next.execute}"
      text="Next"
      disabled="# {!bindings.Next.enabled}"
      partialSubmit="true"/>
    <af:commandButton actionListener="#{bindings.Last.execute}"
      text="Last"
      disabled="# {!bindings.Last.enabled}"
      partialSubmit="true"/>
```

```
</af:panelGroupLayout>
</f:facet>
```

20.4.2.4 Using Automatic Partial Page Rendering

When you create a form with navigational controls, JDeveloper configures the iterator binding to automatically use partial page rendering (PPR). PPR allows only certain components on a page to be rerendered without the need to refresh the entire page. For example, a command link or button can cause another component on the page to be rerendered, without the whole page refreshing. You can configure components to use PPR by setting one component as a target for another component's event, for example a value change or action event. For more information about partial page rendering, see the "Rerendering Partial Page Content" chapter of the *Oracle Fusion Middleware Web User Interface Developer's Guide for Oracle Application Development Framework*.

When you create a form, setting up PPR to work for all the components in the form can be time consuming and error prone. To alleviate this, you can set the `changeEventPolicy` attribute to `ppr` on value bindings. Doing so means that anytime the associated component's value changes as a result of backend business logic, the component will be automatically rerendered. You can also set an iterator binding's `changeEventPolicy` to `ppr`. When you do this, any action or value binding associated with the iterator will act as though its `changeEventPolicy` is set to PPR. This allows entire forms to use PPR without your having to configure each component separately.

When you drop a form and elect to include navigation controls, JDeveloper automatically sets the `changeEventPolicy` attribute on the associated iterator to `ppr`. JDeveloper also sets each of the navigation buttons' `partialSubmit` attribute to `true`. This is how command components notify the framework that PPR should occur. When you set the navigation command components' `partialSubmit` attribute to `true` and you set the iterator's `changeEventPolicy` to `ppr`, each time a navigation button is clicked, all the components in the form that use the iterator binding are rerendered. [Example 20–9](#) shows the page definition code for the iterator created when dropping the `CustomerInfoVO` collection from the Data Controls Panel as a form with navigation.

Example 20–9 ChangeEventPolicy Attribute for an Iterator

```
<iterator Binds="CustomerInfoVO" RangeSize="25"
          DataControl="StoreServiceAMDataControl"
          id="CustomerInfoVOIterator"
          ChangeEventPolicy="ppr" />
```

20.4.3 What Happens at Runtime: How Action Events and Action Listeners Work

An action event occurs when a command component is activated. For example, when a user clicks a **Submit** button, the form is submitted, and subsequently, an action event is fired. Action events might affect only the user interface (for example, a link to change the locale, causing different field prompts to display), or they might involve some logic processing in the back end (for example, a button to navigate to the next record).

An action listener is a class that wants to be notified when a command component fires an action event. An action listener contains an action listener method that processes the action event object passed to it by the command component.

In the case of the navigation operations, when a user clicks, for example, the **Next** button, an action event is fired. This event object stores currency information about the current data object, taken from the iterator. Because the component's `actionListener` attribute is bound to the `execute` method of the **Next** action binding, the **Next** operation is invoked when the event fires. This method takes the currency information passed in the event object to determine what the next data object should be.

In addition, when a user clicks a navigation button, only those components associated with the same iterator as the button's action binding are processed through the lifecycle. This is because a navigation command button's `partialSubmit` attribute is set to `true`, and the associated iterator is configured to use PPR.

20.4.4 What You May Need to Know About the Browser Back Button and Navigating Through Records

You must use the navigation buttons to navigate through the records displayed in a form; you cannot use the browser's back or forward buttons. Because navigation forms automatically use PPR, only part of the page goes through the lifecycle, meaning that when you click a navigation button, the components displaying the data are refreshed and display new data, and you actually remain on the same page. Therefore, when you click the browser's **Back** button, you will be returned to the page that was rendered before the page with the form, instead of to the previous record displayed in the form.

For example, say you are on a page that contains a link to view all current orders. When you click the link, you navigate to a page with a form and the first order, Order #101, is displayed. You then click **Next** and Order #102 is displayed. You click **Next** again, and Order #103 is displayed. If you click the browser's **Back** button, you will not be shown Order #102. Instead, you will be returned to the page that contained the link to view all current orders.

20.5 Creating a Form to Edit an Existing Record

You can create a form that allows a user to edit the current data, and then commit those changes to the data source. To do this, you use operations that can modify data records associated with the collection or the data control itself to create command buttons. For example, you can use the `Delete` operation to create a button that allows a user to delete a record from the current range. Or you can use the built-in **Submit** button to submit changes.

Tip: While you can use the `Create` operation on a form to create a new object, using the ADF Creation Form instead provides additional built-in functionality. See [Section 20.6, "Creating an Input Form"](#) for more information.

If instead of using one of the ADF built-in operations, you want to use a custom method to operate on the data in a form, see [Section 26.2, "Creating Command Components to Execute Methods"](#).

It is important to note that these operations are executed only against objects in the ADF cache. You need to use the `Commit` operation on the root data control to actually commit any changes to the data source. You use the data control's `Rollback` operation to roll back any changes made to the cached object. If the page is part of a transaction within a bounded task flow, you would most likely use these operations to

resolve the transaction in a task flow return activity. For more information, see [Section 17.2, "Managing Transactions"](#).

20.5.1 How to Create Edit Forms

To use the operations on a form, you follow the same procedures as with the navigation operations.

To create an edit form:

- From the Data Controls panel, drag the collection for which you wish to create the form, and select **ADF Form** from the context menu.

This creates a form using `inputText` components, which will allow the user to edit the data in the fields.

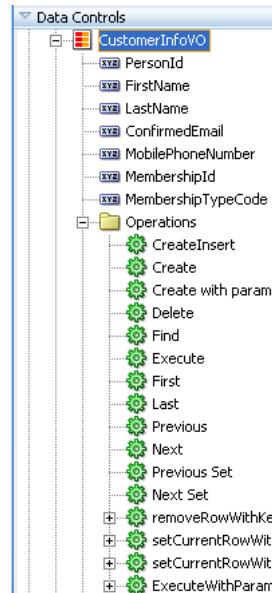
- In the Edit Form Fields dialog, select **Include Submit Button** and click **OK**.
- You can either create new buttons for the operations you want to include, or you can rebind the **Submit** button so that it invokes another operation. To keep the **Submit** button as is and create new buttons for the operations, continue with this step. To rebind the **Submit** button, see Step 5.

From the Data Controls panel, select the operation associated with the collection of objects on which you wish the operation to execute, and drag it onto the JSF page. If you simply want to be able to edit the data, then the **Submit** button is all that is required.

For example, if you want to be able to delete a customer record, you would drag the `Delete` operation associated with the `CustomerInfoVO` collection.

[Figure 20–4](#) shows the operations associated with a collection.

Figure 20–4 Operations Associated with a Collection



- From the ensuing context menu, choose either **ADF Button** or **ADF Link**.
- To rebind the Submit button, right-click the button in the Structure window, and choose **Bind to ADF Control**. In the Bind to ADF Control Dialog, select the

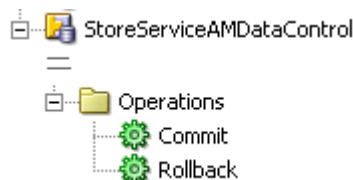
operation to which you want the button bound. Ensure that the operation you select is associated with the collection on which the form is based.

For example, if you want to be able to delete a customer record, you would select the Delete operation associated with the CustomerInfoVO collection.

Figure 20–4 shows the operations associated with a collection.

6. If the page is not part of a transaction within a bounded task flow, then you need to create buttons that allow the user either to commit or roll back the changes. From the Data Controls panel, drag the **Commit** and **Rollback** operations associated with the root-level data control, and drop them as either a command button or a command link. **Figure 20–5** shows the commit and rollback operations for the `StoreServiceAMDataControl` data control (note that for viewing simplicity, the figure omits details in the tree that appear for each view object).

Figure 20–5 Commit and Rollback Operations for a Data Control



If the page is part of a transaction within a bounded task flow, then you can simply enter Commit and Rollback as the values for the transaction resolution when creating the task flow return activity. For more information, see [Section 17.2, "Managing Transactions"](#).

20.5.2 What Happens When You Use Built-in Operations to Change Data

Dropping any data control operation as a command button causes the same events as does dropping navigation operations. For more information, see [Section 20.4.2, "What Happens When You Create Command Buttons"](#).

The only difference is that the action bindings for the Commit and Rollback operations do not require a reference to the iterator, because they execute a method on the application module (the data control itself), as opposed to the iterator. Note that the Rollback action has the `RequiresUpdateModel` property set to `false`. This is because the model should not be updated before the operation is executed, since all changes need to be discarded. [Example 20–10](#) shows the action bindings generated in the page definition file for these operations.

Example 20–10 Action Bindings for Commit and Rollback Operations

```

<action id="Commit" RequiresUpdateModel="true" Action="commitTransaction"
        DataControl="StoreServiceAMDataControl"/>
<action id="Rollback" RequiresUpdateModel="false"
        Action="rollbackTransaction"
        DataControl="StoreServiceAMDataControl"/>
    
```

[Table 20–2](#) shows the built-in non-navigation operations provided on data controls and data control objects, along with the result of invoking the operation or executing an event bound to the operation. For more information about action events, see [Section 20.4.3, "What Happens at Runtime: How Action Events and Action Listeners Work"](#).

Table 20–2 More Built-in Operations

Operation	When invoked, the associated iterator binding will...
CreateInsert	Creates a row directly before the current row, inserts the new record into the row set, then moves the current row pointer to the new row. Note that the range does not move, meaning that the last row in the range may now be excluded from the range. For more information about using the CreateInsert operation to create objects, see Section 20.6, "Creating an Input Form" .
Create	Creates a row directly before the current row, then moves the current row pointer to the new row. Note that the range does not move, meaning that the last row in the range may now be excluded from the range. Also note that the record will not be inserted into the row set, preventing a blank row should the user navigate away without actually creating data. The new row will be created when the user submits the data. For more information, see Section 21.4.4, "What You May Need to Know About Create and CreateInsert" .
CreateWithParameters	Same as the CreateInsert operation (the new record is inserted into the row set), however uses named parameters to create the object.
CreateWithParameters (temporary)	Same as the CreateWithParameters operation, however the newly created record will not be inserted into the row set, preventing a blank row should the user navigate away without actually creating data. The new row will be created when the user submits the data. This difference is the same as the difference between CreateInsert and Create. For more information, see Section 21.4.4, "What You May Need to Know About Create and CreateInsert" .
Delete	Deletes the current row from the cache and moves the current row pointer to the next row in the result set. Note that the range does not move, meaning that a row may be added to the end of the range. If the last row is deleted, the current row pointer moves to the preceding row. If there are no more rows in the collection, the enabled attribute is set to disabled.
RemoveRowByKey	Uses the row key as a String converted from the value specified by the input field to remove the data object in the bound data collection.
Set.CurrentRowWithKey	Sets the row key as a String converted from the value specified by the input field. The row key is used to set the currency of the data object in the bound data collection. For an example of when this is used, see Section 21.2.3, "What You May Need to Know About Setting the Current Row in a Table" .
Set.CurrentRowWithValue	Sets the current object on the iterator, given a key's value. For more information, see Section 21.2.3, "What You May Need to Know About Setting the Current Row in a Table" .
ExecuteWithParams	Refreshes the data collection by first assigning new values to the named bind variables passed as parameters, then (re)executing the view object's query. You would use this operation in the same manner as you would use the CreateInsert operation to create an input form. For more information, see Section 20.6, "Creating an Input Form" . This operation appears only for view objects that have defined one or more named bind variables at design time. For more information, see Section 5.9, "Working with Bind Variables" . For more information about how the page definition works with bind variables and the executeWithParams operation, see Section 26.2.2.1, "Using Parameters in a Method" .
Commit	Causes all items currently in the cache to be committed to the database.
Rollback	Clears the cache and returns the transaction and iterator to the initial state. Resets the ActionListener method.

Table 20–2 (Cont.) More Built-in Operations

Operation	When invoked, the associated iterator binding will...
Execute and Find	These operations are used only in search forms. See Chapter 25, "Creating ADF Databound Search Forms" for more information.

20.6 Creating an Input Form

You can create a form that allows a user to enter information for a new record and then commit that record into the data source. You need to use a task flow that contains a method activity that will call the `CreateInsert` operation before the page with the input form is displayed. This method activity causes a blank row to be inserted into the row set which the user can then populate using a form.

For example, in the StoreFront module, the `customer-registration-task-flow` task flow contains the `createAddress` method activity, which calls the `CreateInsert` operation on the `CustomerAddress` view object. Control is then passed to the `addressDetails` view activity, which displays a form where the user can enter a new address.

Note: If your application does not use task flows, then the calling page should invoke the `createInsert` operation similar to the way in which a task flow's method activity would. For example, you could provide application logic within an event handler associated with a command button on the calling page.

20.6.1 How to Create an Input Form Using a Task Flow

Before you create the input form, you need to create a bounded task flow that will contain both the form and the method activity that will execute the `CreateInsert` operation.

To create an input form:

1. To the bounded task flow, add a method activity. Have this activity execute the `CreateInsert` operation associated with the collection for which you are creating the form. For procedures on using method activities, see [Section 14.5, "Using Method Call Activities"](#).
2. In the Property Inspector, enter a string for the **fixed-outcome** property. For example, the `createAddress` method activity in the `customer-registration-task-flow` task flow has `editAddress` as the **fixed-outcome** value.
3. Add a view activity that represents the page for the input form. For information on adding view activities, see [Section 14.2, "Using View Activities"](#).
4. Add a control flow case from the method activity to the view activity. In the Property Inspector, enter the value of the **fixed-outcome** property of the method activity set in Step 2 as the value of the `from-outcome` of the control flow case.
5. Open the page for the view activity in the design editor, and from the Data Controls panel, drag the collection for which the form will be used to create a new record, and choose **ADF Form** from the context menu.
6. Because you need to commit the new data, the application needs to execute the `commit` operation of the data control. To do this, you can add a button that

navigates to a return activity that calls the commit operation. For procedures for using a return activity, see [Section 14.7, "Using Task Flow Return Activities"](#).

Best Practice: Using the return activity is recommended unless the task flow contains data managed by more than one data control, or you need to commit the data before the end of the flow. In those cases, you can add a button to the page bound to the commit button.

If you need to add a commit button to the page, do the following:

1. In the Data Controls panel, drag the commit operation associated with the data control that contains the collection associated with the input form, and drop it as a command button.
2. In the Structure window, select the command button for the commit operation.
3. In the Property Inspector, set the action to the outcome String that will navigate back to the method activity. You then need to add a control flow case from the page back to the activity, using the same outcome value.
4. Set the command button's disabled property to false.

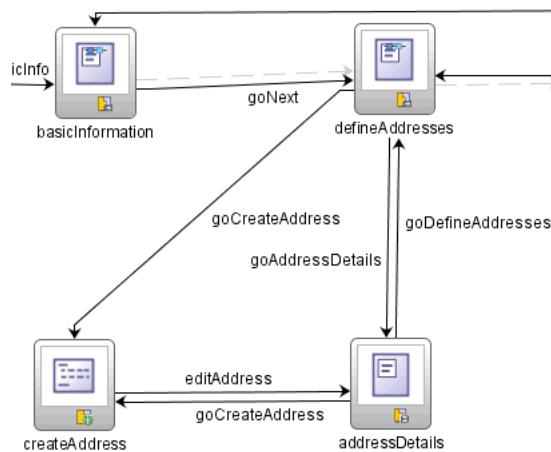
By default, JDeveloper binds the disabled attribute of the button to the enabled property of the binding, causing the button to be disabled when the enabled property is set to false. For this binding, the enabled property is false until an update has been posted. For the purposes of an input form, the button should always be enabled, since there will be no changes posted before the user needs to create the new object.

20.6.2 What Happens When You Create an Input Form Using a Task Flow

When you use an ADF Form to create an input form, JDeveloper:

- Creates an iterator binding for the collection and an action binding for the CreateInsert operation in the page definition for the method activity. The CreateInsert operation is responsible for creating a row in the row set and populating the data source with the entered data. In the page definition for the page, JDeveloper creates an iterator binding for the collection and attribute bindings for each of the attributes of the object in the collection, as for any other form. If you created command buttons or links using the Commit and Rollback operations, JDeveloper also creates an action bindings for those operations.
- Inserts code in the JSF page for the form using ADF Faces `inputText` components, and in the case of the operations, `commandButton` components.

For example, the StoreFront module contains a page that displays all the addresses for a customer in a table. The table includes an **Add** button that navigates to a form where you can input data for a new address. Once the address is created, you return to the page with the table and the new address is displayed. [Figure 20–6](#) shows the `customer-registration-task-flow` task flow with the `createAddress` method activity.

Figure 20–6 Task Flow for an Input Form

[Example 20–11](#) shows the page definition file for the method activity.

Example 20–11 Page Definition Code for a Creation Method Activity

```

<executables>
  <iterator id="CustomerAddressIterator" RangeSize="25"
            Binds="CustomerAddress" DataControl="StoreServiceAMDataControl"/>
</executables>
<bindings>
  <action id="CreateInsert" IterBinding="CustomerAddressIterator"
         InstanceName="StoreServiceAMDataControl.CustomerAddress"
         DataControl="StoreServiceAMDataControl" RequiresUpdateModel="true"
         Action="createInsertRow"/>
</bindings>
  
```

20.6.3 What Happens At Runtime: CreateInsert Action from the Method Activity

When the `createMethodCall` activity is accessed, the `CreateInsert` action binding is invoked, which executes the `CreateInsertRow` operation, and a new blank instance for the collection is created. Note that during routing from the method activity to the view activity, the method activity's binding container skips validation for required attributes, allowing the blank instance to be displayed in the form on the page.

20.6.4 What You May Need to Know About Displaying Sequence Numbers

Because the `Create` action is executed before the page is displayed, if you are populating the primary key using sequences, the next number in the sequence will appear in the input text field, unlike the rest of the fields, which are blank. The sequence number is displayed because the associated entity class contains a method that uses an eager fetch to generate a sequence of numbers for the primary key attribute. The eager fetch populates the value as the row is created. Therefore, using sequences works as expected with input forms.

However, if instead you've configured the attribute's type to `DBSequence` (which uses a database trigger to generate the sequence), the number would not be populated until the object is committed to the database. In this case, the user would see a negative number as a placeholder. To avoid this, you can use the following EL expression for the `Rendered` attribute of the input text field:

```
#(bindings.EmployeeId.inputValue.value > 0}
```

This expression will display the component only when the value is greater than zero, which will not be the case before it is committed. Similarly, you can simply set the Rendered attribute to `false`. However, then the page will never display the input text field component for the primary key.

20.7 Using a Dynamic Form to Determine Data to Display at Runtime

ADF Faces offers a library of dynamic components that includes dynamic form and dynamic table widgets that you can drop from the Data Controls panel. Dynamic components differ from standard components in that all the binding metadata is created at runtime. This dynamic building of the bindings allows you set display information using control hints on a view object. Then if you want to change how the data displays, you need only change it on the view object, and all dynamic components bound to that view object will change their display accordingly. With standard components, if you want to change any display attributes (such as the order or grouping of the attributes) you would need to change each page on which the data is displayed.

For example, in the StoreFront module, you could set the Category and Field Order attribute hints on the `CustomerInfoVO` view object that groups the `FirstName` and `LastName` attributes together and at the top of a form (or at the leftmost columns of a table), the `ConfirmedEmail` and `MobilePhoneNumber` together and at the bottom of a form (or the rightmost columns of a table), and the `MembershipID` and `MembershipType` together and at the middle of a form or table. You could make it so that the `PersonId` does not display at all. For more information about control hints on view objects, see [Section 5.12, "Defining Attribute Control Hints for View Objects"](#).

[Figure 20-7](#) shows a dynamic form at runtime created by dragging and dropping the `CustomerInfoVO` view object with control hints set as described previously, as a dynamic form. Note that the input fields are grayed out because the view object is nonupdateable.

Figure 20-7 Dynamic Form Displays Based on Hints Set on the View Object

First Name	John
Last Name	Chen
Membership Type	PERS
Membership ID	1
Phone	
Email	JCHEN

20.7.1 How to Use Dynamic Forms

To use dynamic forms you first need to set control hints (especially the order and grouping hints) on any corresponding view objects. Next you import the libraries for the dynamic components. You can then drop the dynamic form or table widgets onto your page.

To use dynamic components:

1. Set UI hints on the corresponding view objects. For the Category hint, enter a string that can be used to group attributes together. For example, in the `CustomerInfoVO` hints, the `FirstName` and `LastName` attributes both have `name` as the value for the Category UI hint. The Field Order hint determines the

order the attributes are displayed within a category. For example, in the `CustomerInfoVO` hints, the `FirstName` attribute has a Field Order value of 1 and the `LastName` attribute has a Field Order value of 2.

For procedures on creating UI hints, see [Section 5.12, "Defining Attribute Control Hints for View Objects"](#).

2. If not already included, import the dynamic component library.
 1. In the Application Navigator, right-click the project in which the dynamic components will be used, and from the context menu, choose **Project Properties**.
 2. In the tree, select **JSP Tag Libraries**.
 3. On the JSP Tag Libraries page, click **Add**.
 4. In the Choose Tag Libraries dialog, select **Dynamic Components**, and click **OK**.
 5. On the JSP Tag Libraries page, click **OK**.
3. From the Data Controls panel, select the collection that represents the view object.
4. Drag the collection onto the page, and from the context menu, choose **Forms > ADF Dynamic Form**.

Tip: If dynamic components are not listed, then the library was not imported into the project. Repeat Step 2.

5. In the Property Inspector, for the **Category** field, enter the string used as the value for the Category UI hint for the first group you'd like to display in your form. For example, in [Figure 20–7](#), the Category value would be `name`.
6. Repeat Steps 4 and 5 for each group that you want to display on the form. For example, the form in [Figure 20–7](#) is actually made up of three different forms: one for the category name, one for the category membership, and one for the category contact.

20.7.2 What Happens When You Use Dynamic Components

When you drop a dynamic form, only a binding to the iterator is created.

[Example 20–12](#) shows the page definition for a page that contains one dynamic form component created by dropping the `CustomerInfoVO` collection. Note that no attribute bindings are created.

Example 20–12 Page Definition Code for a Dynamic Form

```
<pageDefinition xmlns="http://xmlns.oracle.com/adfm/uimodel"
    version="11.1.1.46.77" id="DynamicFormPageDef"
    Package="package.pageDefs">
    <parameters/>
    <executables>
        <iterator Binds="CustomerInfoVO" RangeSize="25"
            DataControl="StoreServiceAMDataControl"
            id="CustomerInfoVOIterator"/>
    </executables>
    <bindings/>
</pageDefinition>
```

JDeveloper inserts a `form` tag which contains a dynamic form tag for each of the forms dropped. The `form` tag's value is bound to the iterator binding, as shown in

Example 20–13. This binding means the entire form is bound to the data returned by the iterator. You cannot set display properties for each attribute individuality, nor can you rearrange attributes directly on the JSF page.

Example 20–13 JSF Page Code for a Dynamic Form

```
<af:document>
  <af:messages/>
  <af:form>
    <dynamic:form value="#{bindings.CustomerInfoVOIterator}" category="name"/>
    <dynamic:form value="#{bindings.CustomerInfoVOIterator}" category="member"/>
    <dynamic:form value="#{bindings.CustomerInfoVOIterator}" category="contact"/>
  </af:form>
</af:document>
```

Tip: You can set certain properties that affect the functionality of the form. For example, you can make a form available for upload, set the rendered property, or set a partial trigger. To do this, select the `af:form` tag in the Structure window, and set the needed properties using the Property Inspector.

20.7.3 What Happens at Runtime: How Attribute Values are Dynamically Determined

When a page with dynamic components is rendered, the bindings are created just as they are when items are dropped from the Data Controls panel at design time, except that they are created at runtime. For more information, see [Section 20.3.2, "What Happens When You Create a Form"](#).

Tip: While there is a slight performance hit because the bindings need to be created at runtime, there is also a performance gain because the JSF pages do not need to be regenerated and recompiled when the structure of the view object changes.

20.8 Modifying the UI Components and Bindings on a Form

Once you use the Data Controls panel to create any type of form (except a dynamic form), you can then delete attributes, change the order in which they are displayed, change the component used to display data, and change the attribute to which the components are bound.

Note: You cannot change how a dynamic form displays using the following procedures. You must change display information on the view object or entity object instead.

20.8.1 How to Modify the UI Components and Bindings

You can modify certain aspects of the default components dropped from the Data Controls panel. You can use the Structure window to change the order in which components are displayed, to add new components or change existing components, or to delete components. You can use the Property Inspector to change or delete bindings, or to change the label displayed for a component.

To modify default components and bindings:

1. Use the Structure window to do the following:
 - Change the order of the UI components by dragging them up or down the tree. A black line with an arrowhead denotes where the UI component will be placed.
 - Add a UI component. Right-click an existing UI component in the Structure window and choose to place the new component before, after, or inside the selected component. You then choose from a list of UI components.
 - Bind a UI component. Right-click an existing UI component in the Structure window and choose **Bind to ADF Control**. You can then select the object to which you want your component bound.
 - Rebind a UI component. Right-click an existing UI component in the Structure window and choose **Rebind to another ADF Control**. You can then select the new control object to which you want your component bound.
 - Delete a UI component. Right-click the component and choose **Delete**. If you wish to keep the component, but delete the binding, you need to use the Property Inspector. See the second bullet point in Step 2.
2. With the UI component selected in the Structure window, you can then do the following in the Property Inspector:
 - Add a non-ADF binding for the UI component. Enter an EL expression in the **Value** field, or use the dropdown menu and choose **Edit**.
 - Delete a binding for the UI component by deleting the EL expression.
 - Change the label for the UI component. By default, the label is bound to the binding's `label` property of its hint. This property allows your page to use the UI control hints for labels that you have defined for your entity object attributes or view object attributes. The UI hints allow you to change the value once and have it appear the same on all pages that display the label.

You can change the label just for the current page. To do so, select the `label` attribute. You can enter text or an EL expression to bind the label value to something else, for example, a key in a properties or resource file.

For example, the `inputText` component used to display the name of a product might have the following for its `Label` attribute:

```
#{bindings.ProductName.hints.label}
```

However, you could change the expression to instead bind to a key in a properties file, for example:

```
#{properties['productName']}
```

In this example, `properties` is a variable defined in the JSF page used to load a properties file.

20.8.2 What Happens When You Modify Attributes and Bindings

When you modify how an attribute is displayed by moving or changing the UI component, JDeveloper changes the corresponding code on the JSF page. When you use the binding editors to add or change a binding, JDeveloper adds the code to the JSF page, and also adds the appropriate elements to the page definition file.

Creating ADF Databound Tables

This chapter describes how to use the Data Controls panel to create data bound tables using ADF Faces components.

This chapter includes the following sections:

- [Section 21.1, "Introduction to Adding Tables"](#)
- [Section 21.2, "Creating a Basic Table"](#)
- [Section 21.3, "Creating an Editable Table"](#)
- [Section 21.4, "Creating an Input Table"](#)
- [Section 21.5, "Providing Multiselect Capabilities"](#)
- [Section 21.6, "Modifying the Attributes Displayed in the Table"](#)

21.1 Introduction to Adding Tables

Unlike forms, tables allow you to display more than one data object from a collection at a time. [Figure 21–1](#) shows the Items Ordered tab of the My Orders page in the StoreFront module application, which uses a browse table to display the items for a given order.

Figure 21–1 The Orders Table

ProductId	ProductName
	Zune 30Gb [Audio and Video] Zune is here. Designed around the principles of sharing, discovery and community, Zune new ways for people to connect and share entertainment experiences. The Zune centers around connection- connection to your library, connection to friends, co community and connection to other devices. Zune starts with a 30GB digital media player twist. You can wirelessly share selected full-length ...
	Zune 30Gb [Audio and Video] Zune is here. Designed around the principles of sharing, discovery and community, Zune new ways for people to connect and share entertainment experiences. The Zune centers around connection- connection to your library, connection to friends, co community and connection to other devices. Zune starts with a 30GB digital media player twist. You can wirelessly share selected full-length ...
	Zune 30Gb [Audio and Video] Zune is here. Designed around the principles of sharing, discovery and community, Zune new ways for people to connect and share entertainment experiences. The Zune centers around connection- connection to your library, connection to friends, co community and connection to other devices. Zune starts with a 30GB digital media player twist. You can wirelessly share selected full-length ...

You can create tables that simply display data, or you can create tables that allow you to edit or create data. Once you drop a collection as a table, you can add command

buttons bound to actions that execute some logic on a selected row. You can also modify the default components to suit your needs.

21.2 Creating a Basic Table

Unlike with forms where you bind the individual UI components that make up a form to the individual attributes on the collection, with a table you bind the ADF Faces `table` component to the complete collection or to a range of n data objects at a time from the collection. The individual components used to display the data in the columns are then bound to the attributes. The iterator binding handles displaying the correct data for each object, while the `table` component handles displaying each object in a row. JDeveloper allows you to do this declaratively, so that you don't need to write any code.

21.2.1 How to Create a Basic Table

To create a table using a data control, you bind the `table` component to a collection. JDeveloper allows you to do this declaratively by dragging and dropping a collection from the Data Controls panel.

Tip: You can also create a table by dragging a table component from the Component Palette and completing the Create ADF Faces Table wizard.

To create a databound table:

1. From the Data Controls panel, select a collection.

For example, to create a simple table in the StoreFront module that displays products in the system, you would select the `Products` collection.

2. Drag the collection onto a JSF page, and from the context menu, choose the appropriate table.

When you drag the collection, you can choose from the following types of tables:

- **ADF Table:** Allows you to select the specific attributes you wish your editable table columns to display, and what UI components to use to display the data. By default, ADF `inputText` components are used for most attributes, thus enabling the table to be editable. Attributes that are dates use the `inputDate` component. Additionally, if a control type control hint has been created for an attribute, or if the attribute has been configured to be a list, then the component set by the hint is used instead.
 - **ADF Read-Only Table:** Same as the **ADF Table**; however, each attribute is displayed in an `outputText` component.
 - **ADF Read-Only Dynamic Table:** The attributes returned and displayed are determined dynamically at runtime. This component is helpful when the attributes for the corresponding object are not known until runtime, or you do not wish to hardcode the column names in the JSF page.
 - **ADF Pivot Table:** The pivot table is one of the ADF Data Visualization components. These components provide extensive graphical and tabular capabilities for analyzing data. For more information about using this type of table component, see [Section 24.4, "Creating Databound Pivot Tables"](#).
3. The ensuing Edit Table Columns dialog shows each attribute in the collection, and allows you to determine how these attributes will behave and appear as columns in your table.

Note: If the collection contains a structured attribute (an attribute that is neither a Java primitive type nor a collection), the attributes of the structured attributes will also appear in the dialog.

Using this dialog, you can do the following:

- Allow the ADF Model layer to handle selection by selecting the **Row Selection** checkbox. Selecting this option means that the iterator binding will access the iterator to determine the selected row. Select this option unless you do not want the table to allow selection.
- Allow the ADF Model layer to handle column sorting by selecting the **Sorting** checkbox. Selecting this option means that the iterator binding will access the iterator, which will perform an order-by query to determine the order. Select this option unless you do not want to allow column sorting.
- Allow the columns in the table to be filtered using entered criteria by selecting the **Filtering** checkbox. Selecting this option allows the user to enter criteria in text fields above each column. That criteria is then used to build a Query-by-Example (QBE) search on the collection, so that the table will display only the results returned by the query. For more information, see [Section 25.5, "Creating Filtered Search Tables"](#).
- Group columns for selected attributes together under a parent column, by selecting the desired attributes (shown as rows in the dialog), and clicking the **Group** button. [Figure 21–2](#) shows how three grouped columns appear in the visual editor after the table is created.

Figure 21–2 Grouped Columns in an ADF Faces Table

Group			#...ProductName. label}
#...ProductId.label}	#...SupplierId.label}	#...CategoryId.label}	#...ProductName. label}
#...ProductId)	#...SupplierId)	#...CategoryId)	#...ProductName)
#...ProductId)	#...SupplierId)	#...CategoryId)	#...ProductName)
#...ProductId)	#...SupplierId)	#...CategoryId)	#...ProductName)

- Change the display label for a column. By default, the label is bound to the `label`s property for any control hint defined for the attribute on the table binding. This binding allows you to change the value of a label text once on the view object, and have the change appear the same on all pages that display the label.

In the Edit Table Columns dialog, you can instead enter text or an EL expression to bind the label value to something else, for example, a key in a resource file.

- Change the value binding for a column. You can change the column to be bound to a different attribute. If you simply want to rearrange the columns, you should use the order buttons. If you do change the attribute binding for a column, the label for the column also changes.
- Change the UI component used to display an attribute. The UI components are set based on the table you selected when you dropped the collection onto the page, on the type of the corresponding attribute (for example, `inputDate` components are used for attributes that are dates), and on whether or not

default components were set as control hints on the corresponding view object. You can change to another component using the dropdown menu.

Tip: If you want to use a component that is not listed in the dropdown menu, use this dialog to select the `outputText` component, and then manually add the other tag to the page.

- Change the order of the columns using the order buttons.
 - Add a column using the **Add** icon. There's no limit to the number of columns you can add. When you first click the icon, JDeveloper adds a new column line at the bottom of the dialog and populates it with the values from the first attribute in the bound collection; subsequent new columns are populated with values from the next attribute in the sequence, and so on.
 - Delete a column using the **Delete** icon.
4. Once the table is dropped on the page, you can use the Property Inspector to set other display properties of the table. For example, you may want to set the width of the table to a certain percentage or size. For more information about display properties, see the "Presenting Data in Tables and Trees" chapter of the *Oracle Fusion Middleware Web User Interface Developer's Guide for Oracle Application Development Framework*.
- Tip:** When you set the table width to 100%, the table will not include borders, so the actual width of the table will be larger. To have the table set to 100% of the container width, expand the **Style** section of the Property Inspector, select the **Box** tab, and set the **Border Width** attribute to 0 pixels.
5. If you want the user to be able to edit information in the table and save any changes, you need to provide a way to submit and persist those changes. For more information, see [Section 21.3, "Creating an Editable Table"](#). For procedures on creating tables that allow users to input data, see [Section 21.4, "Creating an Input Table"](#).

21.2.2 What Happens When You Create a Table

Dropping a table from the Data Controls panel has the same effect as dropping a text field or form. Briefly, JDeveloper does the following:

- Creates the bindings for the table and adds the bindings to the page definition file
- Adds the necessary code for the UI components to the JSF page

For more information, see [Section 20.2.2, "What Happens When You Create a Text Field"](#).

21.2.2.1 Iterator and Value Bindings for Tables

When you drop a table from the Data Controls panel, a tree value binding is created. A tree consists of a hierarchy of nodes, where each subnode is a branch off a higher level node. In the case of a table, it is a flattened hierarchy, where each attribute (column) is a subnode off the table. Like an attribute binding used in forms, the tree value binding references the iterator binding, while the iterator binding references an iterator for the data collection, which facilitates iterating over the data objects in the collection.

Instead of creating a separate binding for each attribute, only the tree binding to the table node is created. In the tree binding, the `AttrNames` element within the

The `nodeDefinition` element contains a child element for each attribute that you want to be available for display or reference in each row of the table.

The tree value binding is an instance of the `FacesCtrlHierBinding` class that extends the core `JUCtrlHierBinding` class to add two JSF specific properties: `collectionModel`.

- `collectionModel`: Returns the data wrapped by an object that extends the `javax.faces.model.DataModel` object that JSF and ADF Faces use for collection-valued components like tables.
- `treeModel`: Extends `collectionModel` to return data that is hierarchical in nature. For more information, see [Chapter 22, "Displaying Master-Detail Data"](#).

[Example 21–1](#) shows the value binding for the table created when you drop the `Products` collection.

Example 21–1 Value Binding Entries for a Table in the Page Definition File

```
<bindings>
  <tree IterBinding="ProductsIterator" id="Products">
    <nodeDefinition DefName="oracle.fodemo.storefront.store.queries.ProductsVO">
      <AttrNames>
        <Item Value="ProductId"/>
        <Item Value="SupplierId"/>
        <Item Value="CategoryId"/>
        <Item Value="ProductName"/>
        <Item Value="CostPrice"/>
        <Item Value="ListPrice"/>
        .
        .
        .
      </AttrNames>
    </nodeDefinition>
  </tree>
</bindings>
```

Only the table component needs to be bound to the model (as opposed to the columns or the text components within the individual cells), because only the table needs access to the data. The tree binding for the table drills down to the individual structure attributes in the table, and the table columns can then derive their information from the table component.

21.2.2.2 Code on the JSF Page for an ADF Faces Table

When you use the Data Controls panel to drop a table onto a JSF page, JDeveloper inserts an ADF Faces table component, which contains an ADF Faces `column` component for each attribute named in the table binding. Each column then contains another component (such as an `inputText` or `outputText` component) bound to the attribute's value. Each column's heading is bound to the `label`s property for the control hint of the attribute.

Tip: If an attribute is marked as hidden on the associated view or entity object, no corresponding UI is created for it.

[Example 21–2](#) shows a simplified code excerpt from a table created by dropping the `Products` collection as a read only table.

Example 21–2 Simplified JSF Code for an ADF Faces Table

```

<af:table value="#{bindings.Products.collectionModel}" var="row"
    rows="#{bindings.Products.rangeSize}"
    emptyText="#{bindings.Products.viewable ? 'No rows yet.' :
        'Access Denied.'}"
    fetchSize="#{bindings.Products.rangeSize}"
    selectedRowKeys="#{bindings.Products.collectionModel.selectedRow}"
    selectionListener="#{bindings.Products.collectionModel.makeCurrent}"
    rowSelection="single">
    <af:column sortProperty="ProductId" sortable="true"
        headerText="#{bindings.Products.hints.ProductId.label}">
        <af:outputText value="#{row.ProductId}" />
    </af:column>
    <af:column sortProperty="SupplierId" sortable="true"
        headerText="#{bindings.Products.hints.SupplierId.label}">
        <af:outputText value="#{row.SupplierId}">
            <af:convertNumber groupingUsed="false"
                pattern="#{bindings.Products.hints.SupplierId.format}" />
        </af:outputText>
    </af:column>
    <af:column sortProperty="CostPrice" sortable="true"
        headerText="#{bindings.Products.hints.CostPrice.label}">
        <af:outputText value="#{row.CostPrice}">
            <af:convertNumber groupingUsed="false"
                pattern="#{bindings.Products.hints.CostPrice.format}" />
        </af:outputText>
    </af:column>
    .
    .
    .
</af:table>
```

The tree binding iterates over the data exposed by the iterator binding. Note that the table's value is bound to the collectionModel property, which accesses the collectionModel object. The table wraps the result set from the iterator binding in a collectionModel object. The collectionModel allows each item in the collection to be available within the table component using the var attribute.

In the example, the table iterates over the rows in the current range of the Products iterator binding. The iterator binding binds to a row set iterator that keeps track of the current row. When you set the var attribute on the table to row, each column then accesses the current data object for the current row presented to the table tag using the row variable, as shown for the value of the af:outputText tag:

```
<af:outputText value="#{row.ProductId}" />
```

When you drop an ADF Table (as opposed to an ADF Read Only Table), instead of being bound to the row variable, the value of the input component is implicitly bound to a specific row in the binding container through the bindings property, as shown in [Example 21–3](#). Additionally, JDeveloper adds validator and converter components for each input component. By using the bindings property, any raised exception can be linked to the corresponding binding object or objects. The controller iterates through all exceptions in the binding container and retrieves the binding object to get the client ID when creating FacesMessage objects. This retrieval allows the table to display errors for specific cells. This strategy is used for all input components, including selection components such as lists.

Example 21–3 Using Input Components Adds Validators and Converters

```
<af:table value="#{bindings.Products.collectionModel}" var="row"
    rows="#{bindings.Products.rangeSize}"
    first="#{bindings.Products.rangeStart}"
    emptyText="#{bindings.Products.viewable ? 'No rows yet.' :
        'Access Denied.'}"
    fetchSize="#{bindings.Products.rangeSize}"
    selectedRowKeys="#{bindings.Products.collectionModel.selectedRow}"
    selectionListener="#{bindings.Products.collectionModel.makeCurrent}"
    rowSelection="single">
    <af:column sortProperty="ProductId" sortable="true"
        headerText="#{bindings.Products.hints.ProductId.label}">
        <af:inputText value="#{row.bindings.ProductId.inputValue}"
            <f:validator binding="#{row.bindings.ProductId.validator}"/>
            <af:convertNumber groupingUsed="false"
                pattern="#{bindings.Products.hints.ProductId.format}"/>
        </af:inputText>
    </af:column>
```

For more information about using ADF Faces validators and converters, see the "Validating and Converting Input" chapter of the *Oracle Fusion Middleware Web User Interface Developer's Guide for Oracle Application Development Framework*.

Table 21–1 shows the other attributes defined by default for ADF Faces tables created using the Data Controls panel.

Table 21–1 ADF Faces Table Attributes and Populated Values

Attribute	Description	Default Value
rows	Determines how many rows to display at one time.	An EL expression that, by default, evaluates to the rangeSize property of the associated iterator binding, which determines how many rows of data are fetched from a data control at one time. Note that the value of the rows attribute must be equal to or less than the corresponding iterator's rangeSize value, as the table cannot display more rows than are returned.
first	Index of the first row in a range (based on 0).	An EL expression that evaluates to the rangeStart property of the associated iterator binding.
emptyText	Text to display when there are no rows to return.	An EL expression that evaluates to the viewable property on the iterator. If the table is viewable, the attribute displays No rows yet when no objects are returned. If the table is not viewable (for example, if there are authorization restrictions set against the table), it displays Access Denied .
fetchSize	Number of rows of data fetched from the data source.	An EL expression that, by default, evaluates to the rangeSize property of the associated iterator binding. For more information about the rangeSize property, see Section 20.4.2.2, "Iterator RangeSize Attribute" . This attribute can be set to a larger number than the rows attribute. Note that to improve scrolling behavior in a table, when the table's iterator binding is expected to manage a data set consisting of over 200 items, and the view object is configured to use range paging, the iterator actually returns a set of ranges instead of just one range. For more information about using range paging, see Section 35.1.5, "Efficiently Scrolling Through Large Result Sets Using Range Paging" .

Table 21–1 (Cont.) ADF Faces Table Attributes and Populated Values

Attribute	Description	Default Value
selectedRowKeys	The selection state for the table.	An EL expression that by default, evaluates to the selected row on the collection model.
selectionListener	Reference to a method that listens for a selection event.	An EL expression that by default, evaluates to the makeCurrent method on the collection model.
rowSelection	Determines whether rows are selectable.	Set to single to allow one row to be selected at a time. For information about allowing more than one row to be selected at a time, see Section 21.5, "Providing Multiselect Capabilities" .
Column Attributes		
sortProperty	Determines the property on which to sort the column.	Set to the columns corresponding attribute binding value.
sortable	Determines whether a column can be sorted.	Set to false. When set to true, the iterator binding will access the iterator to determine the order.
headerText	Determines the text displayed at the top of the column.	An EL expression that, by default, evaluates to the label control hint set on the corresponding attribute.

21.2.3 What You May Need to Know About Setting the Current Row in a Table

When you use tables in an application and you allow the ADF Model layer to manage row selection, the current row is determined by the iterator. When a user selects a row in an ADF Faces table, the row in the table is shaded, and the component notifies the iterator of the selected row. To do this, the selectedRowKeys attribute of the table is bound to the collection model's selected row, as shown in [Example 21–4](#).

Example 21–4 Selection Attributes on a Table

```
<aaf:table value="#{bindings.Products1.collectionModel}" var="row"
.
.
.
selectedRowKeys="#{bindings.Products.collectionModel.selectedRow}"
selectionListener="#{bindings.Products.collectionModel.
                           makeCurrent}"
rowSelection="single">
```

This binding binds the selected keys in the table to the selected row of the collection model. The selectionListener attribute is then bound to the collection model's makeCurrent property. This binding makes the selected row of the collection the current row of the iterator.

Note: If you create a custom selection listener, you must create a method binding to the makeCurrent property on the collection model (for example

`# {binding.Products.collectionModel.makeCurrent})` and invoke this method binding in the custom selection listener before any custom logic.

Although a table can handle selection automatically, there may be cases where you need to programmatically set the current row for an object on an iterator.

You can call the `getKey()` method on any view row to get a Key object that encapsulates the one or more key attributes that identify the row. You can also use a Key object to find a view row in a row set using the `findByPrimaryKey()`. At runtime, when either the `setCurrentRowWithKey` or the `setCurrentRowWithValue` built-in operation is invoked by name by the data binding layer, the `findByPrimaryKey()` method is used to find the row based on the value passed in as a parameter before the found row is set as the current row.

The `setCurrentRowWithKey` and `setCurrentRowWithValue` operations both expect a parameter named `rowKey`, but they differ precisely by what each expects that `rowKey` parameter value to be at runtime:

setCurrentRowWithKey

`setCurrentRowWithKey` expects the `rowKey` parameter value to be the *serialized string representation* of a view row key. This is a hexadecimal-encoded string that looks like this:

```
000200000002C20200000002C102000000010000010A5AB7DAD9
```

The serialized string representation of a key encodes all of the key attributes that might comprise a view row's key in a way that can be conveniently passed as a single value in a browser URL string or form parameter. At runtime, if you inadvertently pass a parameter value that is not a legal serialized string key, you may receive exceptions like `oracle.jbo.InvalidParamException` or `java.io.EOFException` as a result. In your web page, you can access the value of the serialized string key of a row by referencing the `rowKeyStr` property of an ADF control binding (for example, `#{bindings.SomeAttrName.rowKeyStr}`) or the `row` variable of an ADF Faces table (e.g. `#{row.rowKeyStr}`).

setCurrentRowWithValue

The `setCurrentRowWithValue` operation expects the `rowKey` parameter value to be the literal value representing the key of the view row. For example, its value would be simply "201" to find product number 201.

Note: If you write custom code in an application module class and need to find a row based on a serialized string key passed from the client, you can use the `getRowFromKey()` method in the `JboUtil` class in the `oracle.jbo.client` package:

```
static public Row getRowFromKey(RowSetIterator rsi, String sKey)
```

The first parameter is the view object instance in which you'd like to find the row. The second parameter is the serialized string format of the key.

21.3 Creating an Editable Table

You can create a table that allows the user to edit information within the table, and then commit those changes to the data source. To do this, you use operations that can modify data records associated with the collection (or the data control itself) to create command buttons, and place those buttons in a toolbar in the table. For example, you might use the `Delete` operation to create a button that allows a user to delete a record from the current range. Or you can use the built-in `Submit` button to submit changes.

Tip: To create a table that allows you to insert a new record into the data store, see [Section 21.4, "Creating an Input Table"](#).

It is important to note that these operations are executed only against objects in the ADF cache. You need to use the `Commit` operation on the root data control to actually commit any changes to the data source. You use the data control's `Rollback` operation to roll back any changes made to the cached object. If the page is part of a transaction within a bounded task flow, you would most likely use these operations to resolve the transaction in a task flow return activity. For more information, see [Section 17.2, "Managing Transactions"](#).

When you decide to use editable components to display your data, you have the option of the table displaying all rows as editable at once, or displaying all rows as read-only until the user double-clicks within the row. For example, [Figure 21–3](#) shows a table whose rows all have editable fields. The page renders using the components that were added to the page (for example, `inputText`, `inputDate`, and `inputComboBoxListOfValues` components).

Figure 21–3 Table With Editable Fields

No.	Name	inputText	* Required field	inputComboBoxListOfValues	inputDate
0	..			7/12/2004	
1	..			7/12/2004	
2	admin.jar	1 KB	1 KB	5/11/2004	
3	applib			7/12/2004	
4	applications			7/12/2004	
5	config			7/12/2004	
6	connectors			7/12/2004	
7	database			7/12/2004	
8	default-web-app			7/12/2004	
9	iiop.jar	1,290 KB	1,290 KB	5/11/2004	
10	iiop_gen_bin.jar	37 KB	37 KB	5/11/2004	
11	iiop_rmic.jar	144 KB	144 KB	5/11/2004	
12	jazn			7/12/2004	

[Figure 21–4](#) shows the same table, but configured so that the user must double-click (or single-click if the row is already selected) a row in order to edit or enter data. Note that `outputText` components are used to display the data in the non-selected rows, even though the same input components as in [Figure 21–3](#) were used to build the page. The only row that actually renders those components is the row selected for editing.

Figure 21–4 Click to Edit a Row

No.	Name	inputText	* Required field	inputComboboxListOf	inputDate
0	..				7/12/2004
1	..				7/12/2004
2	admin.jar	1 KB	1 KB		5/11/2004
3	applib			▼	7/12/2004 
4	applications				7/12/2004
5	config				7/12/2004
6	connectors				7/12/2004
7	database				7/12/2004
8	default-web-app				7/12/2004
9	iiop.jar	1,290 KB	1,290 KB		5/11/2004
10	iiop_gen_bin.jar	37 KB	37 KB		5/11/2004
11	iiop_rmic.jar	144 KB	144 KB		5/11/2004
12	jazn				7/12/2004

For more information about how ADF Faces table components handle editing, see the "Editing Data in Tables, Trees, and Tree Tables" section of the *Oracle Fusion Middleware Web User Interface Developer's Guide for Oracle Application Development Framework*.

21.3.1 How to Create an Editable Table

To create an editable table, you follow similar procedures to creating a basic table, then you add command buttons bound to operations. However, in order for the table to contain a toolbar, you need to add an ADF Faces component that associates the toolbar with the items in the collection used to build the table.

To create an editable table:

1. From the Data Controls panel, select a collection.

For example, to create a simple table in the StoreFront module that will allow you to edit products in the system, you would select the Products collection.

2. Drag the collection onto a JSF page, and from the context menu, choose **ADF Table**.

This creates an editable table using input components.

3. Use the ensuing Edit Table Columns dialog to determine how the attributes should behave and appear as columns in your table. Be sure to select the **Row Selection** checkbox, which will allow the user to select the row to edit.

For more information about using this dialog to configure the table, see [Section 21.2.1, "How to Create a Basic Table"](#).

4. With the table selected in the Structure window, expand the **Behavior** section of the Property Inspector and set the **editingMode** attribute. If you want all the rows to be editable select **editAll**. If you want the user to click into a row to make it editable, select **clickToEdit**.
5. From the Structure window, right-click the table component and select **Surround With** from the context menu.
6. In the Surround With dialog, ensure that **ADF Faces** is selected in the dropdown list, select the **panelCollection** component, and click **OK**.

The **panelCollection** component's toolbar facet will hold the toolbar which, in turn, will hold the command components used to update the data.

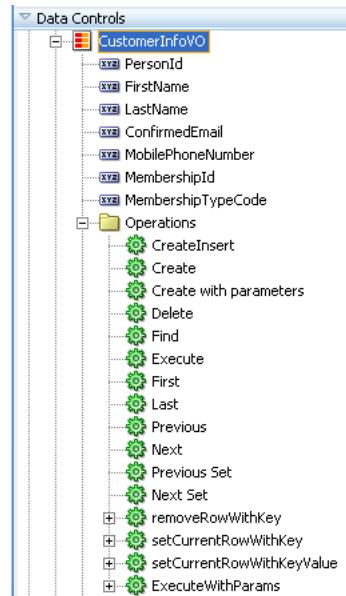
7. In the Structure window, right-click the panelCollection's **toolbar** facet folder and from the context menu, choose **Insert inside toolbar > Toolbar**.

This creates a toolbar that already contains a default menu that allows users to change how the table is displayed and a **Detach** link that detaches the entire table and displays it such that it occupies the majority of the space in the browser window. For more information about the panelCollection component, see the "Displaying Table Menus, Toolbars, and Status Bars" section of the *Oracle Fusion Middleware Web User Interface Developer's Guide for Oracle Application Development Framework*.

8. From the Data Controls panel, select the operation associated with the collection of objects on which you wish the operation to execute, and drag it onto the **toolbar** component in the Structure window. This will place the databound command component inside the toolbar.

For example, if you want to be able to delete a product record, you would drag the **Delete** operation associated with the **Products** collection. [Figure 21–5](#) shows the operations associated with a collection.

Figure 21–5 Operations Associated With a Collection



9. Choose **ADF Toolbar Button** from the context menu.
10. To create a **Submit** button that submits changes to the cache, right-click the toolbar component in the Structure window and choose **Insert inside toolbar > ADF Faces > ADF Toolbar Button**.
11. If the page is not part of a transaction within a bounded task flow, then you need to create buttons that allow the user to either commit or rollback the changes. From the Data Controls panel, drag the Commit and Rollback operations associated with the root-level data control, and drop them as either a command button or command link into the toolbar.

[Figure 21–6](#) shows the commit and roll back operations for the **StoreServiceAMDataControl** data control.

Figure 21–6 Commit and Rollback Operations for a Data Control

If the page is part of a transaction within a bounded task flow, then you can simply enter Commit and Rollback as the values for the transaction resolution when creating the task flow return activity. For more information, see [Section 17.2, "Managing Transactions"](#).

21.3.2 What Happens When You Create an Editable Table

Creating an editable table is similar to creating a form used to edit records. Action bindings are created for the operations dropped from the Data Controls panel. For details on what happens when you create an editable table, see [Section 20.4.2, "What Happens When You Create Command Buttons"](#).

21.4 Creating an Input Table

You can create a table that allows users to insert a new blank row into a table and then add values for each column (any default values set on the corresponding entity or view object will be automatically populated).

21.4.1 How to Create an Input Table

When you create an input table, you want the user to see the new blank row in the context of the other rows within the current row set. To allow this insertion, you need to use the `CreateInsert` operation instead of the `Create` operation (as you would use with forms). The `CreateInsert` operation actually creates the new row within the row set instead of only in the cache.

ADF Faces components can be set so that one component refreshes based on an interaction with another component, without the whole page needing to be refreshed. This is known as *partial page rendering*. Because the table needs to refresh when the user clicks a button to create the new row, you need to configure the table to respond to that user action.

Before you begin, create an editable table, as described in [Section 21.3, "Creating an Editable Table"](#). If your table is not part of a bounded task flow, be sure to include buttons bound to the `Commit` and `Rollback` operations.

To create an input table:

1. From the Data Controls panel, drag the `CreateInsert` operation associated with the dropped collection and drop it as a toolbar button into the toolbar. Because this is the component that when activated needs to refresh the table, enter a unique ID for the button.
2. In the Structure window, select the table component. In the Property Inspector, expand the **Behavior** section.
3. In the Property Inspector, click the dropdown menu for the **PartialTriggers** attribute, and select **Edit**.
4. In the Edit Property dialog, expand the facet for the `panelCollection` component that contains the `CreateInsert` command component. Select that

component (based on the ID you gave it in Step 1) and shuttle it to the **Selected** panel. Click **OK**. This sets that component to be the trigger that will cause the table to refresh.

21.4.2 What Happens When You Create an Input Table

When you use the `CreateInsert` operation to create an input table, JDeveloper:

- Creates an iterator binding for the collection, an action binding for the `CreateInsert` operation, and attribute bindings for the table. The `CreateInsert` operation is responsible for creating the new row in the row set. If you created command buttons or links using the `Commit` and `Rollback` operations, JDeveloper also creates an action bindings for those operations.
- Inserts code in the JSF page for the table using ADF Faces `table`, `column`, and `inputText` components, and in the case of the operations, `commandButton` components.

[Example 21–5](#) shows the page definition file for an input table created from the `Products` collection (some attributes were deleted in the Edit Columns dialog when the collection was dropped).

Example 21–5 Page Definition Code for an Input Table

```
<executables>
    <iterator Binds="Products" RangeSize="25"
        DataControl="StoreServiceAMDataControl" id="ProductsIterator" />
</executables>
<bindings>
    <tree IterBinding="ProductsIterator" id="Products">
        <nodeDefinition DefName="oracle.fodemo.storefront.store.queries.ProductsVO">
            <AttrNames>
                <Item Value="ProductId"/>
                <Item Value="ProductName"/>
                <Item Value="CostPrice"/>
                <Item Value="ListPrice"/>
                <Item Value="Description"/>
                <Item Value="CategoryName"/>
                <Item Value="CategoryDescription"/>
                <Item Value="ProductImageId"/>
            </AttrNames>
        </nodeDefinition>
    </tree>
    <action IterBinding="ProductsIterator" id="CreateInsert"
        RequiresUpdateModel="true" Action="createInsertRow" />
    <action id="Commit" RequiresUpdateModel="true" Action="commitTransaction"
        DataControl="StoreServiceAMDataControl" />
    <action id="Rollback" RequiresUpdateModel="false"
        Action="rollbackTransaction"
        DataControl="StoreServiceAMDataControl" />
</bindings>
```

[Example 21–6](#) shows the code added to the JSF page that provides partial page rendering, using the `CreateInsert` command toolbar button as the trigger to refresh the table.

Example 21–6 Partial Page Trigger Set on a Command Button for a Table

```
<af:form>
    <af:panelCollection>
```

```

<f:facet name="menus"/>
<f:facet name="toolbar">
  <af:toolbar>
    <af:commandToolbarButton actionListener="#{bindings.CreateInsert.execute}"
      text="CreateInsert"
      disabled="#{!bindings.CreateInsert.enabled}"
      id="CreateInsert"/>
    <af:commandToolbarButton actionListener="#{bindings.Commit.execute}"
      text="Commit"
      disabled="false"/>
    <af:commandToolbarButton actionListener="#{bindings.Rollback.execute}"
      text="Rollback"
      disabled="#{!bindings.Rollback.enabled}"
      immediate="true">
      <af:resetActionListener/>
    </af:commandToolbarButton>
  </af:toolbar>
</f:facet>
<f:facet name="statusbar"/>
<af:table value="#{bindings.Products.collectionModel}" var="row"
  rows="#{bindings.Products.rangeSize}"
  emptyText="#{bindings.Products.viewable} ? \'No rows yet.\' :
  \'Access Denied.\'"
  fetchSize="#{bindings.Products.rangeSize}"
  rowSelection="single" partialTriggers="CreateInsert">
  <af:column sortProperty="ProductId" sortable="false"
    headerText="#{bindings.Products.hints.ProductId.label}">
    <af:inputText value="#{row.ProductId}" simple="true"
      required="#{bindings.Products.hints.ProductId.mandatory}"
      columns="#{bindings.Products.hints.ProductId.displayWidth}"
      maximumLength="#{bindings.Products.hints.
        productId.precision}"/>
  </af:column>
  .
  .
  .
  </af:table>
</af:panelCollection>
</af:form>

```

21.4.3 What Happens at Runtime: How CreateInsert and Partial Page Refresh Work

When the button bound to the `CreateInsert` operation is invoked, the action executes, and a new instance for the collection is created and inserted as the page is rerendered. Because the button was configured to be a trigger that causes the table to refresh, the table redraws with the new empty row shown at the top. When the user clicks the button bound to the `Commit` action, the newly created rows in the rowset are inserted into the database. For more information about partial page refresh, see the "Refreshing Partial Page Content" chapter in the *Oracle Fusion Middleware Web User Interface Developer's Guide for Oracle Application Development Framework*.

21.4.4 What You May Need to Know About Create and CreateInsert

When you use the `Create` or `CreateInsert` operation to declaratively create a new row, it performs the following lines of code:

```

// create a new row for the view object
Row newRow = yourViewObject.createRow();
// mark the row as being "initialized", but not yet new

```

```
newRow.setNewRowState(Row.STATUS_INITIALIZED);
```

However, if you are using the `CreateInsert` operation, it performs the additional line of code to insert the row into the row set:

```
// insert the new row into the view object's default rowset  
yourViewObject.insertRow(newRow);
```

When you create a row in an entity-based view object, the `Transaction` object associated with the current application module immediately takes note of the fact. The new entity row that gets created behind the view row is already part of the `Transaction`'s list of pending changes. When a newly created row is marked as having the *initialized* state, it is removed from the `Transaction`'s pending changes list and is considered a blank row in which the end user has not yet entered any data values. The term *initialized* is appropriate since the end user will see the new row initialized with any default values that the underlying entity object has defined. If the user never enters any data into any attribute of that initialized row, then it is as if the row never existed. At transaction commit time, since that row is not part of the `Transaction`'s pending changes list, no `INSERT` statement will be attempted for it.

As soon as at least one attribute in an initialized row is set, it automatically transitions from the initialized status to the new status (`Row.STATUS_NEW`). At that time, the underlying entity row is enrolled in the `Transaction`'s list of pending changes, and the new row will be permanently saved the next time you commit the transaction.

Note: If the end user performs steps that result in creating many initialized rows but never populating them, it might seem like a recipe for a slow memory leak. However, the memory used by an initialized row that never transitions to the new state will eventually be reclaimed by the Java virtual machine's garbage collector.

21.5 Providing Multiselect Capabilities

By default, when you drop a table component and set it to use row selection, it is set to allow a user to select a single row. You can change the table so that using the CTRL key or the SHIFT key, the user can select multiple rows, allowing the application to work on multiple rows at the same time.

In order to allow the user to select and operate on more than one row, among other things, you need to change the `rowSelection` attribute to `multiple`. In Fusion web applications, operations (such as methods) work on the current data object, which the iterator keeps track of. When the `rowSelection` attribute is set to `single` (as it is by default when you select the **Row Selection** checkbox in the Edit Table Columns dialog when creating the table), the table is able to show the current data object as being selected, and it is also able to set any newly selected row to the current object on the iterator. If the same iterator is used on a subsequent page (for example, if the user selects a row and then clicks the command button to navigate to a page where the object can be edited), the selected object will be displayed. This selection and navigation works because the iterator and the component are working with a single object: the notion of the current row is the same because the different iterator bindings in different binding containers are bound to the same row set iterator.

However, when you set the `rowSelection` attribute to `multiple`, there potentially could be multiple selected objects. The ADF Model layer has no notion of "selected" as opposed to "current." You must add logic to the model layer that takes the component instance, reads the selected rows, and translates the selection state to the binding. The method can then perform some action on the selected rows.

21.5.1 How to Add Multiselect Capabilities

To add multiselect capabilities, you first modify certain table component attributes. Then you use a managed bean to handle setting the selected rows as the current rows.

Tip: You can follow the same procedures for adding multiselect capabilities to a tree or tree table.

Before you begin, create a table as described in [Section 21.2, "Creating a Basic Table"](#), being sure to NOT select the **Row Selection** checkbox.

Note: You should NOT select the **Row Selection** checkbox, because when you do, JDeveloper automatically binds the `selectionListener` attribute to the `makeCurrent` method on the `CollectionModel` class and the `selectedRowKeys` attribute to the `selectedRow` property on that class. Both are designed to work only for single selection tables. Make sure that for any table for which you need multiple selection, that neither of these attributes are populated.

To add multiselect capabilities:

1. In the Structure window, select the table component and set the following attributes in the Property Inspector:
 - **Row Selection:** `multiple`
 - **ID:** an ID of your choice for the table
2. Create and register a managed bean for the page, if one does not already exist. You'll use the managed bean to handle the row selection and execute logic against the selection (for example, deleting the selected rows), so that it executes against all selected rows. For procedures for creating a managed bean, see [Section 18.4, "Using a Managed Bean in a Fusion Web Application"](#).
3. Add a property to the managed bean that represents the table component. Set the value of the property to the table ID value as set in Step 1, and the property class to be the ADF Faces table component class, as shown in [Example 21–7](#).

Example 21–7 Property on a Managed Bean That Represents a Table

```
<managed-property>
  <property-name>table1</property-name>
  <property-class>oracle.adf.view.rich.component.rich.data.RichTable
    </property-class>
  <value>#{table1}</value>
</managed-property>
```

4. Add getter and setter methods for the table to the managed bean, as shown in [Example 21–8](#).

Example 21–8 Getter and Setter Methods for the Table Component

```
private RichTable _table1;

public void setTable1(RichTable table1) {
    this.table1 = table1;
}
public RichTable getTable1() {
    return table1;
```

```
}
```

Tip: You'll need to import the ADF Faces table class into the managed bean. JDeveloper can do this for you declaratively when you press CTRL+ENTER after adding the code in [Example 21–8](#).

5. Back in the JSP page, in the Structure window select the table component and set the binding attribute to be bound to the managed property you created in Step 3. For example:

```
binding="#{myBean.table1}"
```

This binding is what allows the application to work with the entire table component. For more information about the binding attribute and how it allows you access objects programmatically, refer to the Java EE 5 tutorial on Sun's web site (<http://java.sun.com>)

Tip: If you elected to use automatic component binding when creating the JSF page, then most of Steps 2 through 5 will have been done for you. You need only create the set method on the managed bean, as shown in Step 6. It is important, however, to understand the behavior of the page when using automatic component binding. For more information, see the "What You May Need to Know About Automatic Component Binding" section of the *Oracle Fusion Middleware Web User Interface Developer's Guide for Oracle Application Development Framework*.

6. Add logic to allow the declarative operation or method to operate against the set of selected rows. The additional logic you add needs to do the following:

- Look up the binding in the page definition file.
- Get a hold of the component instance by creating a property binding of the component's binding property.
- From the instance, read the selected rows (or in the case of trees or tree tables, the selected nodes and leaves).
- Translate the selection state to the underlying binding's row key to obtain the ADF binding references for the selected rows.
- Execute the logic on the rows, (for example, to delete rows, get the binding reference and remove the selected rows from the collection model).

[Example 21–9](#) shows the method named `deleteOnTable()` that removes the selected rows from the address table on the `updateUserInfo` page.

Example 21–9 Deleting Multiple Rows on an Iterator

```
public void deleteOnTable(String bName,RichTable myTable ) {  
    FacesContext fctx = FacesContext.getCurrentInstance();  
    Application application = fctx.getApplication();  
    ELContext elctx = fctx.getELContext();  
    ExpressionFactory exprfactory = application.getExpressionFactory();  
  
    ValueExpression valueExpression =  
        exprfactory.createValueExpression(elctx,"#{bindings}",Object.class);  
  
    DCBindingContainer dcbinding =
```

```
(DCBindingContainer) valueExpression.getValue(elctx);

FacesCtrlHierBinding treeRootNode =
    (FacesCtrlHierBinding) dcbinding.get(bName);

RowKeySet rowKeySet = (RowKeySet) myTable.getSelectedRowKeys();
CollectionModel cm = treeRootNode.getCollectionModel();
for (Object facesTreeRowKey : rowKeySet) {
    cm.setRowKey(facesTreeRowKey);
    JUCtrlHierNodeBinding rowData = (JUCtrlHierNodeBinding)
cm.getRowData();
    rowData.getRow().remove();
}

}
```

21.5.2 What Happens at Runtime: How an Operation Executes Against Multiple Rows

When the user selects multiple rows and then clicks the command button, the application uses different contexts to access the expression factory to build an expression that resolves to the binding container. It then retrieves the iterator and the selected row keys on the component, and sets the selected rows on the binding. Next, it uses the collection model of the binding to remove the row and then reaccesses the component to display the collection with the now removed rows.

21.6 Modifying the Attributes Displayed in the Table

Once you use the Data Controls panel to create a table, you can then delete attributes, change the order in which they are displayed, change the component used to display them, and change the attribute binding for the component. You can also add new attributes, or rebind the table to a new data control.

21.6.1 How to Modify the Displayed Attributes

You can modify the following aspects of a table that was created using the Data Controls panel:

- Change the binding for the label of a column
 - Change the attribute to which a UI component is bound
 - Change the UI component bound to an attribute
 - Reorder the columns in the table
 - Delete a column in the table
 - Add a column to the table
 - Enable selection and sorting

To change the attributes for a table:

1. In the Structure window, select the table component.
 2. In the Property Inspector, expand the different sections to change the attributes for the table.

21.6.2 How to Change the Binding for a Table

Instead of modifying a binding, you can completely change the object to which the table is bound.

To rebind a table:

1. Right-click the table in the Structure window and choose **Rebind to Another ADF Control**.
2. In the Bind to ADF Control dialog, select the new collection to which you want to bind the table. Note that changing the binding for the table will also change the binding for all the columns. You can then use the procedures in [Section 21.6.1, "How to Modify the Displayed Attributes"](#) to modify those bindings.

Tip: You can also rebind a table by dragging a different view object on top of the existing table.

21.6.3 What Happens When You Modify Bindings or Displayed Attributes

When you simply modify how an attribute is displayed by moving the UI component or changing the UI component, JDeveloper changes the corresponding code on the JSF page. When you use the binding editors to add or change a binding, JDeveloper adds the code to the JSF page, and also adds the appropriate elements to the page definition file.

22

Displaying Master-Detail Data

This chapter describes how to create various types of pages that display master-detail related data.

This chapter includes the following sections:

- [Section 22.1, "Introduction to Displaying Master-Detail Data"](#)
- [Section 22.2, "Identifying Master-Detail Objects on the Data Controls Panel"](#)
- [Section 22.3, "Using Tables and Forms to Display Master-Detail Objects"](#)
- [Section 22.4, "Using Trees to Display Master-Detail Objects"](#)
- [Section 22.5, "Using Tree Tables to Display Master-Detail Objects"](#)

For information about using a selection list to populate a collection with a key value from a related master or detail collection, see [Chapter 23, "Creating Databound Selection Lists and Shuttles"](#).

22.1 Introduction to Displaying Master-Detail Data

In ADF Business Components, a master-detail relationship refers to two view object instances that are related by a view link. As described in [Section 5.1, "Introduction to View Objects"](#), 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. ADF uses the view link to associate a row of one view object instance (the master object) with one or more rows of another view object instance (the detail object).

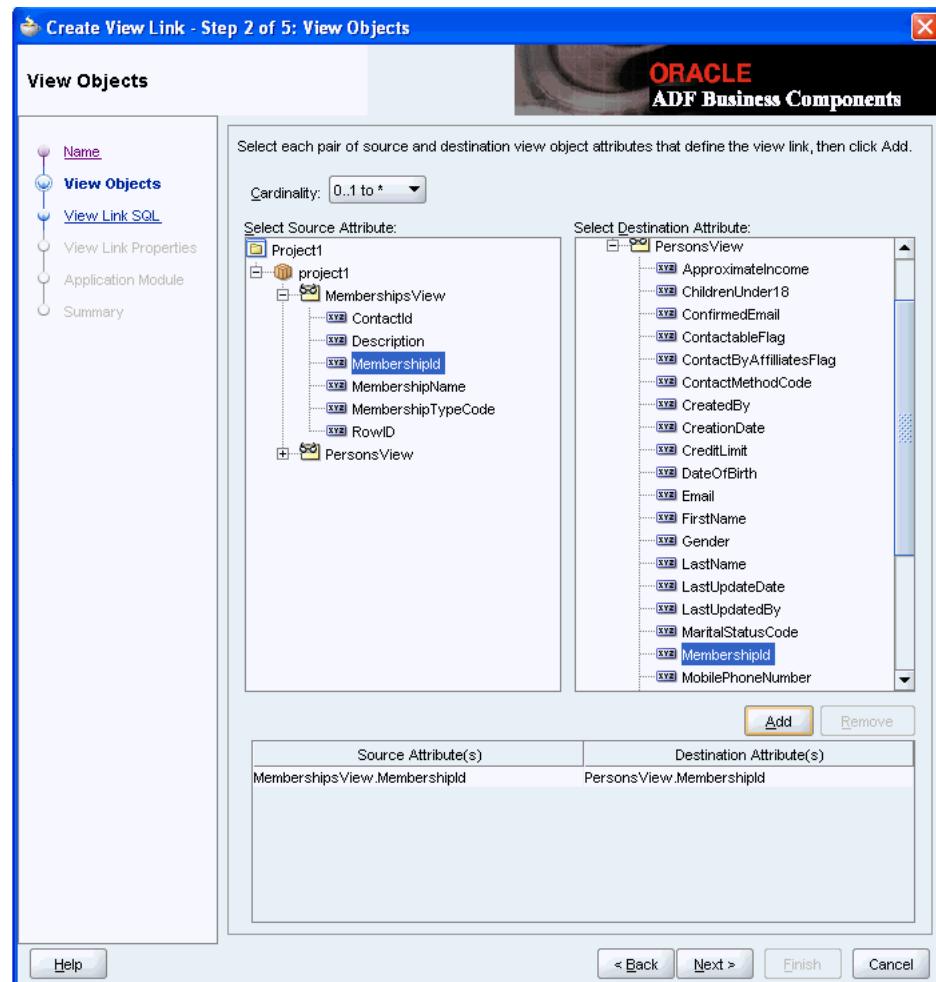
View links support two different types of master-detail coordination: view link accessor attributes and data model view link instances (for more information, see [Section 35.1.3, "Understanding View Link Accessors Versus Data Model View Link Instances"](#)). However, when displaying master-detail data on a page using ADF data binding, you exclusively use data model view link instances, which support master-detail coordination.

When you use ADF Business Components, in combination with the ADF Model layer and ADF Faces UI components, the data model will automatically update to reflect any changes to the rowsets of these business objects (for more information, see [Section 3.4, "Overview of the UI-Aware Data Model"](#)).

To enable UI-Aware data model master-detail coordination, you must add both the master view object and the detail view object instances to the application module data model (for more information, see [Section 5.6.4, "How to Enable Active Master-Detail Coordination in the Data Model"](#)). For example, in the Fusion Order Demo application, there is a view link from the Memberships view object to the Persons view object

based on the MembershipId attribute. Both the master and detail view objects have been added to the application module data model. So, a change in the current row of the master view object instance causes the rowset of the detail view object instance to refresh to include the details for the current master.

Figure 22–1 View Link between Memberships and Persons View Objects



When objects have a master-detail relationship, you can declaratively create pages that display the data from both objects simultaneously. For example, the page shown in [Figure 22–2](#) displays a Country Code in a form at the top of the page and its related States and Provinces in a table at the bottom of the page. This is possible because the service request and service history objects have a master-detail relationship. In this example, the Country Code is the master object and States and History is the detail object. ADF iterators automatically manage the synchronization of the detail data objects displayed for a selected master data object. Iterator bindings simplify building user interfaces that allow scrolling and paging through collections of data and drilling-down from summary to detail information.

Figure 22–2 Detail Table

The screenshot shows a user interface for managing data. At the top, there is a header labeled "Country Codes" with a sub-header "CountryId US". Below this is a toolbar with buttons for "First", "Previous", "Next", and "Last". The main area contains a table titled "States and Provinces". The table has two columns: "CountryId" and "StateProvince". The data in the table is as follows:

CountryId	StateProvince
US	NV
US	AZ
US	MO
US	IN
US	CA
US	MI
US	NE
US	VA
US	SC
US	TX
US	WI
US	PA
US	NY
US	OR
US	FL

You display master and detail objects in forms and tables. The master detail form can display these objects on separate pages. For example, JDeveloper options allow you to display the master object in a table on one page and detail objects in a read-only form on another page.

Note: There are some cases when the master-detail UI components that JDeveloper provides cannot provide the functionality you require. For example, you may need to bind components by hand instead of using the master-detail UI components.

A master object can have many detail objects, and each detail object can have its own detail objects up to many levels of depth. If one of the detail objects in this hierarchy is dropped from the Application Navigator as a master-detail form on a page, only its immediate parent master object displays on the page. It will not display all the way up to the topmost parent object.

If you display the detail object as a tree or tree table object, it is possible to display the entire hierarchy with multiple levels of depth, starting with the topmost master object, and traversing detail children objects at each node.

22.2 Identifying Master-Detail Objects on the Data Controls Panel

You can declaratively create pages that display master-detail data using the Data Controls panel. The Data Controls panel displays master-detail related objects in a hierarchy that mirrors the one you defined in the application module data model, where the detail objects are children of the master objects. For information about adding master-detail objects to the data model, see [Section 5.6.4, "How to Enable Active Master-Detail Coordination in the Data Model"](#).

To display master-detail objects as form or table objects, drag the detail object from the data control in the Application navigator and drop it on the page. Its master object is automatically created on the page.

Figure 22–3 shows two master-detail related collections in the Data Controls panel of the Fusion Order Demo application. The PersonsView collection is an instance of the Persons view object, and the MembershipsView collection, which appears as a child of the PersonsView collection, is an instance of the Memeberships view object. The master-detail hierarchy on the Data Controls panel reflects the hierarchy defined in the StoreFront application module data model, as shown in [Figure 22–2](#). The hierarchy was established by creating a view link from the Persons view object to the Memberships view object. Next, an instance of the resulting detail view object, MembershipsView1 via PersonMembershipLink1, has been added to the application module data model, as shown in [Figure 22–4](#).

Tip: The master-detail hierarchy displayed in the Data Controls panel does not reflect the cardinality of the relationship (for example, one-to-many, one-to-one, many-to-many). The hierarchy simply shows which collection (the master) is being used to retrieve one or more objects from another collection (the detail).

Figure 22–3 Master-Detail Objects in the Data Controls Panel

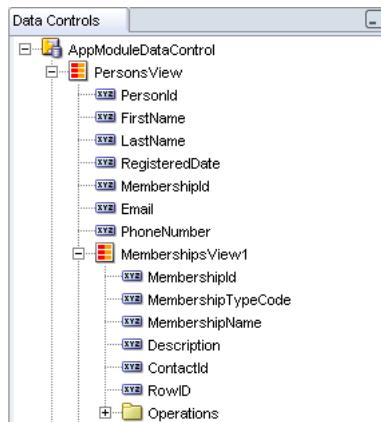
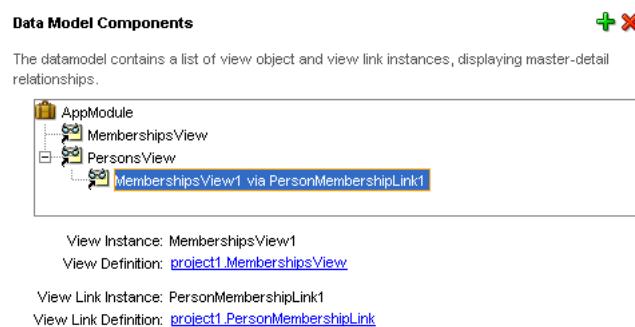


Figure 22–4 Master-Detail Hierarchy Defined in the Application Module Data Model



In the Fusion Order Demo application, the view link between the Persons view object and Memberships view object is a one-way relationship. If the view link was bi-directional and both sets of master and detail view objects were added to the application module data model, then the Data Controls panel would also display the MembershipsView collection at the same node level as the PersonsView collection,

and the detail instance of the `PersonsView` collection as child of the `MembershipsView` collection.

When creating a page that displays master-detail objects, be sure to correctly identify which object is the master and which is the detail for your particular purposes. Otherwise, you may not display the desired data on the page.

For example, if you want to display a user and all the related expertise areas to which the user is assigned, then `User` would be the master object. However, if you wanted to display an expertise area and all the users assigned to it, then `expertiseArea` would be the master object. The detail objects displayed on a page depend on which object is the master.

Tip: By default, when you define a view link using the Create View Link wizard, the source view object is the master and the destination view object is the detail. However, if you choose to generate accessors in both the source and the destination view objects, then the master-detail relationship is bi-directional. If both sets of master-detail view objects resulting from a bi-directional view link are added to the application module data model, then instances of both sets of view objects will appear independently on the Data Controls panel.

For more information about the icons displayed on the Data Controls panel, see ["Section 11.3.1, "How to Use the Data Controls Panel"](#).

22.3 Using Tables and Forms to Display Master-Detail Objects

You can create a master-detail browse page in a single declarative action using the Data Controls pane. You do not need to write any extra code. Even the navigation is included. The Data Controls panel provides pre-built master-detail widgets that display both the master and detail objects on the same page as any combination of read-only tables and forms. All you have to do is drop the detail collection on the page and choose the type of widget you want to use.

The pre-built master-detail widgets available from the Data Controls panel include range navigation that enables the end user to scroll through the data objects in collections. The table provided by the pre-built master-detail widgets includes a selection facet and **Submit** command button. By default, all attributes of the master and detail objects are included in the master-detail widgets as text fields (in forms) or columns (in tables). You can delete unwanted attributes by removing the text field or column from the page.

Tip: If you do not want to use the pre-built master-detail widgets, you can drag and drop the master and detail objects individually as tables and forms on a single page or on separate pages. For more information about creating individual forms and tables, see [Section 20.3, "Creating a Basic Form"](#) or [Section 21.2, "Creating a Basic Table"](#).

When you add master-detail components to a page, the iterator bindings are responsible for exposing data to the components on the page. The iterator bindings bind to the underlying rowset iterators. At runtime, the UI-aware data model and the rowset iterator for the detail view object instance keep the rowset of the detail view object refreshed to the correct set of rows for the current master row as that current row changes.

Figure 22–5 shows an example of pre-built master-detail widget, which displays customer information in a form at the top of the page and an address in a table at the bottom of the page. When the user clicks the **Next** button to scroll through customer names in the master data at the top of the page, the page automatically displays the related detail data, the customer's address.

Figure 22–5 Pre-Built Data Controls Panel Master-Detail Widget

The screenshot displays a pre-built master-detail widget. At the top, there is a form titled "Person" containing the following data:

PrincipalName	SKING
FirstName	Steven
LastName	King
PersonTypeCode	STAFF
PrimaryAddressId	1

Below the form are navigation buttons: a left arrow, a right arrow, a "Next" button, and a "Last" button. The "Next" button is highlighted. Below these buttons is a table titled "Address" with one row of data:

Address1	City	PostalCode	StateProvince	CountryId
514 W Superior St	Kokomo	46901	IN	US

22.3.1 How to Display Master-Detail Objects in Tables and Forms

The Data Controls panel enables you to create both the master and detail widgets on one page with a single declarative action using pre-built master-detail forms and tables. For information about displaying master and detail data on separate pages, see [Section 22.3.4, "What You May Need to Know About Master-Detail on Separate Pages"](#).

To create a master-detail page using the pre-built ADF master-detail forms and tables:

1. From the Data Controls panel, locate the detail object, as described in [Section 22.2, "Identifying Master-Detail Objects on the Data Controls Panel"](#).
2. Drag and drop the detail object onto the JSF page.

Note: If you want to create an editable master-detail form, drop the master object and the detail object separately on the page.

3. In the context menu, choose one of the following **Master-Details** UI components:

- **ADF Master Table, Detail Form:** Displays the master objects in a table and the detail objects in a read-only form under the table.

When a specific data object is selected in the master table, the first related detail data object is displayed in the form below it. The user must use the form navigation to scroll through each subsequent detail data objects.

- **ADF Master Form, Detail Table:** Displays the master objects in a read-only form and the detail objects in a read-only table under the form.

When a specific master data object is displayed in the form, the related detail data objects are displayed in a table below it.

- **ADF Master Form, Detail Form:** Displays the master and detail objects in separate forms.

When a specific master data object is displayed in the top form, the first related detail data object is displayed in the form below it. The user must use the form navigation to scroll through each subsequent detail data object.

- **ADF Master Table, Detail Table:** Displays the master and detail objects in separate tables.

When a specific master data object is selected in the top table, the first set of related detail data objects are displayed in the table below it.

If you want to modify the default forms or tables, see [Section 20.3, "Creating a Basic Form"](#) or [Section 21.2, "Creating a Basic Table"](#).

22.3.2 What Happens When You Create Master-Detail Tables and Forms

When you drag and drop a collection from the Data Controls panel, JDeveloper does many things for you, including adding code to the JSF page and corresponding entries in the page definition file. For a full description of what happens and what is created when you use the Data Controls panel, see [Section 11.3.1, "How to Use the Data Controls Panel"](#).

22.3.2.1 Code Generated in the JSF Page

The JSF code generated for a pre-built master-detail widget is basically the same as the JSF code generated when you use the Data Controls panel to create a basic read-only form or table. If you are building your own master-detail widgets, you might want to consider including similar components that are automatically included in the pre-built master-detail tables and forms.

The tables and forms in the pre-built master-detail widgets include a `panelHeader` tag that contains the fully qualified name of the data object populating the form or table. You can change this label as needed using a string or an EL expression that binds to a resource bundle.

If there is more than one data object in a collection, a form in a pre-built master-detail widget includes four `commandButton` tags for range navigation: `First`, `Previous`, `Next`, and `Last`. These range navigation buttons enable the user to scroll through the data objects in the collection. The `actionListener` of each button is bound to a data control operation, which performs the navigation. The `execute` property used in the `actionListener` binding, invokes the operation when the button is clicked. (If the form displays a single data object, JDeveloper would automatically omit the range navigation components.) For more information about range navigation, see [Section 20.4, "Incorporating Range Navigation into Forms"](#).

Tip: If you drop an **ADF Master Table**, **Detail Form** or **ADF Master Table**, **Detail Table** widget on the page, the parent tag of the detail component (for example, panelForm tag or table tag) automatically has the partialTriggers attribute set to the id of the master component (see [Section 21.4.1, "How to Create an Input Table"](#) for more information about partial triggers). At runtime, the partialTriggers attribute causes only the detail component to be re-rendered when the user makes a selection in the master component, which is called partial rendering. When the master component is a table, ADF uses partial rendering, because the table does not need to be re-rendered when the user simply makes a selection in the facet: only the detail component needs to be re-rendered to display the new data.

22.3.2.2 Binding Objects Defined in the Page Definition File

[Example 22-1](#) shows the page definition file created for a master-detail page that was created by dropping the ServiceHistories collection, which is a detail object under the ServiceRequests object, on the page as an **ADF Master Form**, **Detail Table**.

The executables element defines two iterators: one for the person (which is the master object) and one for the address (which is the detail object). The underlying view link from the master view object to the detail view object establishes the relationship between the two iterators. At runtime, the UI-aware data model and the rowset iterator for the detail view object instance keep the rowset of the detail view object refreshed to the correct set of rows for the current master row as that current row changes (for more information, see [Section 22.3.3, "What Happens at Runtime"](#)).

The bindings element defines the value bindings for the form and the table. The attribute bindings that populate the text fields in the form are defined in the attributeValues elements. The id attribute of the attributeValues element contains the name of each data attribute, and the IterBinding attribute references an iterator binding to display data from the master object in the text fields.

The attribute bindings that populate the text fields in the form are defined in the attributeValues elements. The id attribute of the attributeValues element contains the name of each data attribute, and the IterBinding attribute references an iterator binding to display data from the master object in the text fields.

The range navigation buttons in the form are bound to the action bindings defined in the action elements. As in the attribute bindings, the IterBinding attribute of the action binding references the iterator binding for the master object.

The table, which displays the detail data, is bound to the table binding object defined in the table element. The IterBinding attribute references the iterator binding for the detail object.

For more information about the elements and attributes of the page definition file, see [Section A.7, "pageNamePageDef.xml"](#).

Example 22-1 Binding Objects Defined in the Page Definition for a Master-Detail Page

```
<executables>
    <iterator Binds="PersonsView" RangeSize="10"
        DataControl=" AppModuleDataControl" id="PersonsViewIterator" />
    <iterator Binds="AddressesView" RangeSize="10"
        DataControl=" AppModuleDataControl" id="AddressesViewIterator" />
</executables>
<bindings>
    <attributeValues IterBinding="PersonsViewIterator" id="PersonId">
```

```

<AttrNames>
    <Item Value="PersonId" />
</AttrNames>
</attributeValues>
<attributeValues IterBinding="PersonsViewIterator" id="PrincipalName">
    <AttrNames>
        <Item Value="PrincipalName" />
    </AttrNames>
</attributeValues>
<attributeValues IterBinding="PersonsViewIterator" id="FirstName">
    <AttrNames>
        <Item Value="FirstName" />
    </AttrNames>
</attributeValues>
<attributeValues IterBinding="PersonsViewIterator" id="LastName">
    <AttrNames>
        <Item Value="LastName" />
    </AttrNames>
</attributeValues>
<action id="First" RequiresUpdateModel="true" Action="12"
    IterBinding="PersonsViewIterator"/>
<action id="Previous" RequiresUpdateModel="true" Action="11"
    IterBinding="PersonsViewIterator"/>
<action id="Next" RequiresUpdateModel="true" Action="10"
    IterBinding="PersonsViewIterator"/>
<action id="Last" RequiresUpdateModel="true" Action="13"
    IterBinding="PersonsViewIterator"/>
    <table id="ServiceRequestsServiceHistories"
        IterBinding="ServiceHistoriesIterator">
</table>
</bindings>

```

22.3.3 What Happens at Runtime

At run-time, an ADF iterator determines which row from the master table object to display in the master-detail form. When the form first displays, the first master table object row appears highlighted in the master section of the form. Detail table rows that are associated with the master row display in the detail section of the form.

As described in [Section 22.3.2.2, "Binding Objects Defined in the Page Definition File"](#), ADF iterators are associated with underlying RowSetIterator objects. These manage which data objects, or *rows*, currently display on a page. At runtime, the rowset iterators manage the data displayed in the master and detail components.

Both the master and detail rowset iterators listen to rowset navigation events, such as the user click the range navigation buttons, and display the appropriate row in the UI. In the case of the default master-detail components, the rowset navigation events are the command buttons on a form (**First**, **Previous**, **Next**, **Last**).

The rowset iterator for the detail collection manages the synchronization of the detail data with the master data. Because of the underlying view link from the master view object to the detail view object, the detail rowset iterator listens for row navigation events in both the master and detail collections. If a rowset navigation event occurs in the master collection, the detail rowset iterator automatically executes and returns the detail rows related to the current master row.

22.3.4 What You May Need to Know About Master-Detail on Separate Pages

The default master-detail components display the master-detail data on a single page. However, using the master and detail objects on the Data Controls panel, you can also display the collections on separate pages, and still have the binding iterators manage the synchronization of the master and detail objects.

For example, in the Fusion Order Demo application, the product table and product details are displayed on the Home page. However, the page could display the product list only. Instead of showing the product list, it could provide a button called **Details**. If the user clicks the **Details** button, the application would navigate to a new page that displays all the related details in a form. A button on the service history page would enable the user to return to the service request page.

To display master-detail objects on separate pages, create two pages, one for the master object and one for the detail object, using the individual tables or forms available from the Data Controls panel. Remember that the detail object iterator manages the synchronization of the master and detail data. So, be sure to drag the appropriate detail object from the Data Controls panel when you create the page to display the detail data (see [Section 22.2, "Identifying Master-Detail Objects on the Data Controls Panel"](#)).

To handle the page navigation, create an ADF task flow, then add two view activities to it, one for the master page and one for the detail page. (See [Section 13.2, "Creating Task Flows"](#) for information about associating a View activity with an existing JSF page.) Add command buttons or links to each page, or use the default **Submit** button available when you create a form or table using the Data Controls panel. Each button must specify a navigation rule outcome value in the `action` attribute. In the `task-flow-defintion.xml` file, add a navigation rule from the master data page to the detail data page, and another rule to return from the detail data page to the master data page. The `from-outcome` value in the navigation rules must match the outcome value specified in the `action` attribute of the buttons. For information about adding navigation between pages, see [Section 13.2, "Creating Task Flows"](#).

22.4 Using Trees to Display Master-Detail Objects

In addition to tables and forms, you can also display master-detail data in hierarchical trees. The ADF Faces `tree` component is used to display hierarchical data. It can display multiple root nodes that are populated by a binding on a master object. Each root node in the tree may have any number of branches, which are populated by bindings on detail objects. A tree can have multiple levels of nodes, each representing a detail object of the parent node. Each node in the tree is indented to show its level in the hierarchy.

The `tree` component includes mechanisms for expanding and collapsing the tree nodes; however, it does not have focusing capability. If you need to use focusing, consider using the ADF Faces `TreeTable` component (for more information, see [Section 22.5, "Using Tree Tables to Display Master-Detail Objects"](#)). By default, the icon for each node in the tree is a folder; however, you can use your own icons for each level of nodes in the hierarchy.

[Figure 22–6](#) shows an example of a tree located on the `home.jsp` page of the Fusion Order Demo application. The tree displays two levels of nodes: root and branch. The root node displays parent product categories such as Media, Office, and Electronics. The branch nodes display and subcategories under each parent category, such as Hardware, Supplies, and Software under the Media parent category.

Figure 22–6 Databound ADF Faces Tree

22.4.1 How to Display Master-Detail Objects in Trees

A tree consists of a hierarchy of nodes, where each subnode is a branch off a higher level node. Each node level in a databound ADF Faces tree is populated by a different data collection. In JDeveloper, you define a databound tree using the Tree Binding Editor, which enables you to define the rules for populating each node level in the tree. There must be one rule for each node level in the hierarchy. Each rule defines the following node level properties:

- The data collection that populates that node level
- The attributes from the data collection that are displayed at that node level
- A view link accessor attribute that returns a detail object to be displayed as a branch of the current node level (for information about view link accessors, see [Chapter 5.6.6.2, "Programmatically Accessing a Detail Collection Using the View Link Accessor"](#))

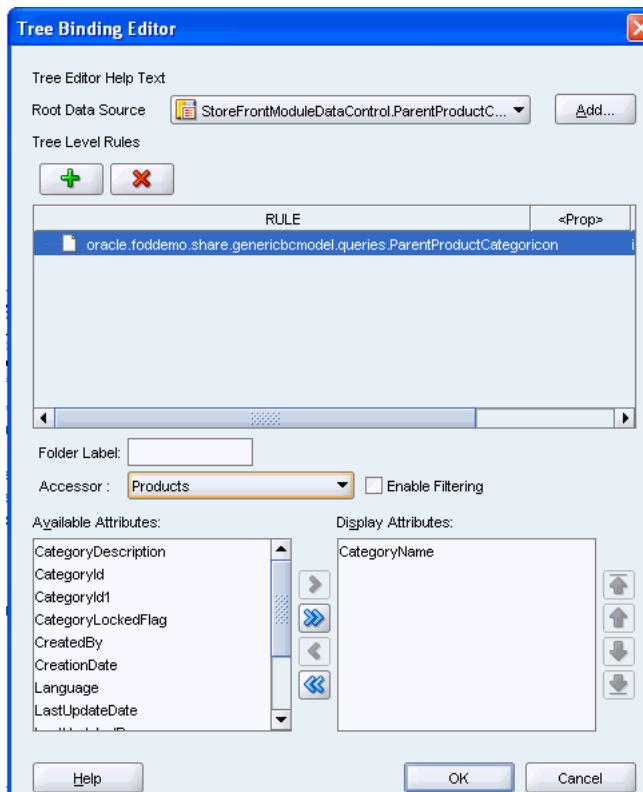
To create the Browse tree on the Fusion Order Demo home page shown in [Figure 22–6](#), a view object, `ParentProductCategories` was created to return a list of parent categories. Another view object, `ProductCategories` was created to return subcategories of products. A view link was created from the `ParentProductCategories` view object to the `ProductCategories` view object, thus establishing a master-detail relationship.

To add a third-level node, for example, a list of products under each subcategory, a view link would need to exist from the `ProductCategories` object to the `Products` view object. For more information about creating view links, see [Section 5.6.6, "How to Access the Detail Collection Using the View Link Accessor"](#).

To display master-detail objects in a tree:

1. Drag the master object from the Data Controls panel, and drop it onto the page. This should be the master data that will represent the root level of the tree.
2. In the context menu, choose **Trees > ADF Tree**.

JDeveloper displays the Tree Binding Editor, as shown in [Figure 22–7](#). You use the Tree Binding Editor to define a rule for each level that you want to appear in the tree.

Figure 22–7 Tree Binding Editor

3. In the **Root Data Source** list, select the data collection that will populate the root node level. This will be the master data collection. By default, this is the same collection that you dragged from the Data Controls panel to create the tree, which was a master collection.

Tip: If you don't see the data collection you want In the **Root Data Source** list, click the **Add** button. In the Add Data Source dialog, select a data control and an iterator name to create a new data source.

4. Click the + icon to add the Root Data Source you selected to the **Tree Level Rules** list.
5. In the **Tree Level Rules** list, select the data source you just added.
6. In the **Accessor** list, select a view link accessor attribute.

The list displays only the accessor attributes that return the detail collections for the master collection you selected. For example, if you are defining the ParentProductCategories node level and you want to add a detail level that displays all subcategories under each parent category, you would select the accessor attribute that returns the ProductCategories collection.

If you select <none>, the node will not expand to display any detail collections, thus ending the branch.

View link accessor attributes, which return data collections, are generated when you create a view link. The **Accessor** field displays all accessor attributes that return detail collections for the master collection selected in the **Tree Level Rules** list. For more information about view objects, view links, and view link accessors,

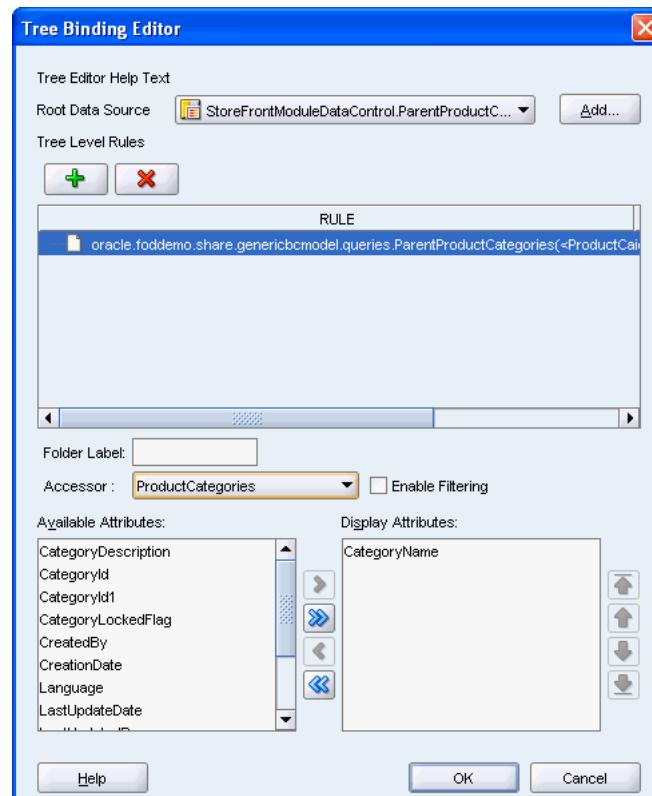
see [Chapter 5.6.6, "How to Access the Detail Collection Using the View Link Accessor"](#).

7. Select an attribute in the **Available Attributes** list and move it to the **Display Attributes** list.

The attribute will be used to display at nodes at the master level. For example, for the Parent Product Categories level, you might select the **Categories** attribute.

When you are finished, the Tree Binding Editor should look contain values similar to those in [Figure 22–8](#).

Figure 22–8 Tree Binding Editor with Master Tree Level Rule Selections

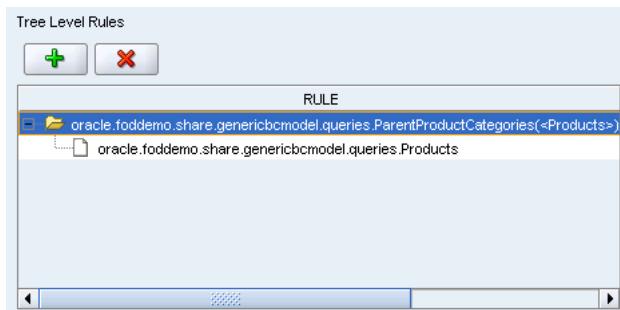


After defining a rule for the master level, you must next define a second rule for the detail level that will appear under the master level in the tree.

For example, in the sample tree shown in [Figure 22–6](#), the first rule added to the Tree Binding editor populates the parent category nodes (Media, Office, and Electronics). The detail level rule populates the product category nodes (for example, Hardware, Supplies, and Software under the Media parent category).

8. To add a second rule, click the Click the + icon above the **Tree Level Rules** list.

A detail data source should appear automatically under the master data source as shown in [Figure 22–9](#).

Figure 22–9 Tree Level Rules

For example, if you specified `ParentProductCategories` as the master Root Data Source, `ProductCategories` will automatically appear underneath in the **Tree Level Rules** list, because the two data sources share a master-detail relationship.

9. Click **OK** and save your work.
10. You can add data sources to the **Tree Level Rules** list. to increase the number of nodes that display in the tree. The order of the remaining data sources should follow the hierarchy of the nodes you want to display in the tree.

22.4.2 What Happens When You Create ADF Databound Trees

When you drag and drop from the Data Controls panel, JDeveloper does many things for you. For a full description of what happens and what is created when you use the Data Controls panel, see [Section 11.3.1, "How to Use the Data Controls Panel"](#).

When you create a databound tree using the Data Controls panel, JDeveloper adds binding objects to the page definition file, and it also adds the tree tag to the JSF Page. The resulting UI component is fully functional and does not require any further modification.

22.4.2.1 Code Generated in the JSF Page

[Example 22–2](#) shows the code generated in a JSF page when you use the Data Controls panel to create a tree. This sample tree displays two levels of nodes: parent product categories and product categories. The `ParentProductCategories` collection was used to populate the root node.

Example 22–2 Code Generated in the JSF Page for a Databound Tree

```
<h:form binding="#{backing_tree1.form1}" id="form1">
    <af:tree value="#{bindings.ParentProductCategories.treeModel}"
        var="node" binding="#{backing_tree1.tree1}" id="tree1">
        <f:facet name="nodeStamp">
            <af:outputText value="#{node}"
                binding="#{backing_tree1.outputText}"
                id="outputText"/>
        </f:facet>
    </af:tree>
</h:form>
```

By default, the `af:tree` tag is created inside a form. The `value` attribute of the tree tag contains an EL expression that binds the tree component to the `ParentProductCategories` tree binding object in the page definition file. The

`treeModel` property in the binding expression refers to an ADF class that defines how the tree hierarchy is displayed, based on the underlying data model. The `var` attribute provides access to the current node.

In the `f:facet` tag, the `nodeStamp` facet is used to display the data for each node. Instead of having a component for each node, the tree repeatedly renders the `nodeStamp` facet, similar to the way rows are rendered for the ADF Faces table component.

The ADF Faces tree component uses an instance of the `oracle.adf.view.faces.model.PathSet` class to display expanded nodes. This instance is stored as the `treeState` attribute on the component. You may use this instance to programmatically control the expanded or collapsed state of an element in the hierarchy. Any element contained by the `PathSet` instance is deemed expanded. All other elements are collapsed.

22.4.2.2 Binding Objects Defined in the Page Definition File

[Example 22–3](#) shows the binding objects defined in the page definition file for the ADF databound tree.

Example 22–3 Binding Objects Defined in the Page Definition File for a Databound Tree

```
<executables>
    <iterator Binds="ParentProductCategories1" RangeSize="10"
        DataControl="StoreFrontModuleDataControl"
        id="ParentProductCategories1Iterator"/>
</executables>
<bindings>
    <tree IterBinding="ParentProductCategories1Iterator"
        id="ParentProductCategories1">
        <AttrNames>
            <Item Value="CategoryName"/>
            <Item Value="CategoryDescription"/>
            <Item Value="Language"/>
            <Item Value="SourceLang"/>
            <Item Value="CategoryId1"/>
            <Item Value="CategoryId"/>
            <Item Value="ParentCategoryId"/>
            <Item Value="CategoryLockedFlag"/>
            <Item Value="RepresentativeProductId"/>
            <Item Value="CreatedBy"/>
            <Item Value="CreationDate"/>
            <Item Value="LastUpdatedBy"/>
            <Item Value="LastUpdateDate"/>
            <Item Value="ObjectVersionId"/>
        </AttrNames>
        <nodeDefinition
            DefName="oracle.foddemo.share.genericcbcmodel.queries.ParentProductCategories">
            <AttrNames>
                <Item Value="CategoryName"/>
            </AttrNames>
            <Accessors>
                <Item Value="ProductCategories"/>
            </Accessors>
        </nodeDefinition>
        <nodeDefinition
            DefName="oracle.foddemo.share.genericcbcmodel.queries.ProductCategories">
            <AttrNames>
                <Item Value="CategoryName"/>
            </AttrNames>
        </nodeDefinition>
    </tree>
</bindings>
```

```
</AttrNames>
</nodeDefinition>
</tree>
</bindings>
```

The page definition file contains the rule information defined in the Tree Binding Editor. In the `executables` element, notice that although the tree displays two levels of nodes, only one iterator binding object is needed. This iterator iterates over the master collection, which populates the root nodes of the tree. The accessor you specified in the node rules return the detail data for each branch node.

The `tree` element is the value binding for all the attributes displayed in the tree. The `iterBinding` attribute of the `tree` element references the iterator binding that populates the data in the tree. The `AttrNames` element within the `tree` element defines binding objects for *all* the attributes in the master collection. However, the attributes that you select to appear in the tree are defined in the `AttrNames` elements within the `nodeDefinition` elements.

The `nodeDefinition` elements define the rules for populating the nodes of the tree. There is one `nodeDefinition` element for each node, and each one contains the following attributes and subelements:

- `DefName`: An attribute that contains the fully qualified name of the data collection that will be used to populate the node.
- `id`: An attribute that defines the name of the node.
- `AttrNames`: A subelement that defines the attributes that will be displayed in the node at runtime.
- `Accessors`: A subelement that defines the accessor attribute that returns the next branch of the tree.

The order of the `nodeDefinition` elements within the page definition file defines the order or level of the nodes in the tree, were the first `nodeDefinition` element defines the root node. Each subsequent `nodeDefinition` element defines a sub-node of the one before it.

For more information about the elements and attributes of the page definition file, see [Appendix A.7, "pageNamePageDef.xml"](#).

22.4.3 What Happens at Runtime

Tree components use `org.apache.myfaces.trinidad.model.TreeModel` to access data. This class extends `CollectionModel`, which is used by the ADF Faces table component to access data. For more information about the `TreeModel` class, refer to the ADF Faces Javadoc.

When a page with a tree is displayed, the iterator binding on the tree populates the root nodes. When a user collapses or expands a node to display or hide its branches, a `DisclosureEvent` event is sent. The `isExpanded` method on this event determines whether the user is expanding or collapsing the node. The `DisclosureEvent` event has an associated listener.

The `DisclosureListener` attribute on the tree is bound to the `accessor` attribute specified in the node rule defined in the page definition file. This `accessor` attribute is invoked in response to the `DisclosureEvent` event; in other words, whenever a user expands the node the `accessor` attribute populates the branch nodes.

22.5 Using Tree Tables to Display Master-Detail Objects

Use the ADF Faces `treeTable` component to display a hierarchy of master-detail collections in a table. The advantage of using a `treeTable` component rather than a `tree` component is that the `treeTable` component provides a mechanism that enables users to focus the view on a particular node in the tree.

[Figure 22–10](#) shows an example of a tree table that displays three levels of nodes: countries, states or provinces, and cities. Each root node represents an individual country. The branches off the root nodes display the state or provinces in the country. Each state or province node branches to display the cities contained in it.

As with trees, to create a tree table with multiple nodes, it is necessary to create view links between the view objects. The view links establish the master-detail relationships. For example, to create the tree table shown in [Figure 22–10](#), it was necessary to create view links from the `CountryCodes` object to the `StatesandProvinces` view object, and another view link from the `StatesandProvinces` view object to the `Cities` view object. For more information about creating view links, see [Section 5.6, "Working with Multiple Tables in a Master-Detail Hierarchy"](#).

Figure 22–10 Databound ADF Faces Tree Table

The screenshot shows a treeTable component with the following data structure:

Category	Item
Media	1
Media	5
Media	10
Media	12
Media	15
Media	16
Media	29
Media	18
Media	19
Media	20
Media	21
Media	22
Office	
Electronics	
Audio and Video	

A databound ADF Faces `treeTable` displays one root node at a time, but provides navigation for scrolling through the different root nodes. Each root node can display any number of branch nodes. Every node is displayed in a separate row of the table, and each row provides a focusing mechanism in the leftmost column.

You can edit the following `treeTable` component properties in the Property Inspector:

- Range navigation: The user can click the **Previous** and **Next** navigation buttons to scroll through the root nodes.
- List navigation: The list navigation, which is located between the **Previous** and **Next** buttons, enables the user to navigate to a specific root node in the data collection using a selection list.
- Node expanding and collapsing mechanism: The user can open or close each node individually or use the **Expand All** or **Collapse All** command links. By default, the icon for opening closing the individual nodes is an arrowhead with a plus or minus sign. You can also use a custom icon of your choosing.
- Focusing mechanism: When the user clicks on the focusing icon (which is displayed in the leftmost column) next to a node, the page is redisplayed showing only that node and its branches. A navigation link is provided to enable the user to return to the parent node.

22.5.1 How to Display Master-Detail Objects in Tree Tables

The steps for creating an ADF Faces databound tree table are exactly the same as those for creating an ADF Faces databound tree, except that you drop the data collection as an **ADF Tree Table** instead of an **ADF Tree**.

22.5.2 What Happens When You Create a Databound Tree Table

When you drag and drop from the Data Controls panel, JDeveloper does many things for you. For a full description of what happens and what is created when you use the Data Controls panel, see [Section 11.3.1, "How to Use the Data Controls Panel"](#).

When you create a databound tree table using the Data Controls panel, JDeveloper adds binding objects to the page definition file, and it also adds the `treeTable` tag to the JSF Page. The resulting UI component is fully functional and does not require any further modification.

22.5.2.1 Code Generated in the JSF Page

[Example 22–4](#) shows the code generated in a JSF page when you use the Data Controls panel to create a tree table. This sample tree table displays three levels of nodes: users, service requests, and service history.

By default, the `treeTable` tag is created inside a form. The `value` attribute of the tree table tag contains an EL expression that binds the tree component to the binding object that will populate it with data, which in the example is the `LoggedInUser` tree binding object. The `treeModel` property refers to an ADF class that defines how the tree hierarchy is displayed, based on the underlying data model. The `var` attribute provides access to the current node.

Example 22–4 Code Generated in the JSF Page for a Databound ADF Faces Tree Table

```
<af:form binding="#{backing_tree3.form1}" id="form1">
    <af:treeTable value="#{bindings.ProductCategories1.treeModel}"
                  var="node" binding="#{backing_tree3.treeTable1}"
                  id="treeTable1">
        <f:facet name="nodeStamp">
            <af:column>
                <af:outputText value="#{node}" />
            </af:column>
        </f:facet>
        <f:facet name="pathStamp">
            <af:outputText value="#{node}"
                           binding="#{backing_tree3.outputText1}"
                           id="outputText1" />
        </f:facet>
    </af:treeTable>
</af:form>
<h:form>
```

In the `facet` tag, the `nodeStamp` facet is used to display the data for each node. Instead of having a component for each node, the tree repeatedly renders the `nodeStamp` facet, similar to the way rows are rendered for the ADF Faces table component. The `pathStamp` facet renders the column and the path links above the table that enable the user to return to the parent node after focusing on a detail node.

22.5.2.2 Binding Objects Defined in the Page Definition File

The binding objects created in the page definition file for a tree table are exactly the same as those created for a tree.

22.5.3 What Happens at Runtime

Tree components use `oracle.adf.view.faces.model.TreeModel` to access data. This class extends `CollectionModel`, which is used by the ADF Faces `table` component to access data. For more information about the `TreeModel` class, refer to the ADF Faces Javadoc.

When a page with a tree table is displayed, the iterator binding on the `treeTable` component populates the root node and listens for a row navigation event (such as the user clicking the **Next** or **Previous** buttons or selecting a row from the range navigator). When the user initiates a row navigation event, the iterator displays the appropriate row.

If the user changes the view focus (by clicking on the component's focus icon), the `treeTable` component generates a focus event (`FocusEvent`). The node to which the user wants to change focus is made the current node before the event is delivered. The `treeTable` component then modifies the `focusPath` property accordingly. You can bind the `FocusListener` attribute on the tree to a method on a managed bean. This method will then be invoked in response to the focus event.

When a user collapses or expands a node, a disclosure event (`DisclosureEvent`) is sent. The `isExpanded` method on the disclosure event determines whether the user is expanding or collapsing the node. The disclosure event has an associated listener, `DisclosureListener`. The `DisclosureListener` attribute on the tree table is bound to the accessor attribute specified in the node rule defined in the page definition file. This accessor attribute is invoked in response to a disclosure event (for example, the user expands a node) and returns the collection that populates that node.

The `treeTable` component includes **Expand All** and **Collapse All** links. When a user clicks one of these links, the `treeTable` sends a `DisclosureAllEvent` event. The `isExpandAll` method on this event determines whether the user is expanding or collapsing all the nodes. The table then expands or collapses the nodes that are children of the root node currently in focus. In large trees, the expand all command will not expand nodes beyond the immediate children. The ADF Faces `treeTable` component uses an instance of the `oracle.adf.view.faces.model.PathSet` class to determine expanded nodes. This instance is stored as the `treeState` attribute on the component. You can use this instance to programmatically control the expanded or collapsed state of a node in the hierarchy. Any node contained by the `PathSet` instance is deemed expanded. All other nodes are collapsed. This class also supports operations like `addAll()` and `removeAll()`.

Like the ADF Faces `table` component, a `treeTable` component provides for range navigation. However, instead of using the `rows` attribute, the `treeTable` component uses a `rowsByDepth` attribute whose value is a space-separated list of non-negative numbers. Each number defines the range size for a node level on the tree. The first number is the root node of the tree, and the last number is for the branch nodes. If there are more branches in the tree than numbers in the `rowsByDepth` attribute, the tree uses the last number in the list for the remaining branches. Each number defines the limit on the number items displayed at one time in each branch. If you want to display all items in a branch, specify 0 in that position of the list.

For example, if the `rowsByDepth` attribute is set to 0 0 3, all root nodes will be displayed, all direct children of the root nodes will be displayed, but only three nodes will display per branch after that. The `treeTable` component includes links to navigate to additional nodes, enabling the user to display the additional nodes.

For more information about the ADF Faces `TreeTable` component, refer to the `oracle.adf.view.faces.component.core.data.CoreTreeTable` class in the ADF Faces Javadoc.

22.5.4 Using the TargetIterator Property

You can expand a node binding in the page definition editor to view the page's node Definition elements. These are the same tree binding rules that you can configure in the tree binding dialog.

For each node definition (rule), you can specify an optional TargetIterator property. Its value is an EL expression that is evaluated at runtime when the user selects a row in the tree. The EL expression evaluates an iterator binding in the current binding container. The iterator binding's view row key attributes match (in order, number, and datatype) the view row key of the iterator from which the nodeDefinition type's rows are retrieved for the tree.

At run time, when the tree control receives a selectionChanged event, it passes in the list of keys for each level of the tree. These uniquely identify the selected node.

The tree binding starts at the top of the tree. For each tree level whose key is present in the Currently Selected Tree Node Keys List, if there is a TargetIterator property configured for that nodeDefinition, the tree binding performs a setCurrentRowWithKey() operation on the selected target iterator. It uses the key from the appropriate level of the Currently Selected Tree Node Keys List.

For example, you may have created DeptEO and EmpEO entity objects, and created view links based on these entity objects. The view link accessor name in the DeptVO object that returns the linked collection of employees in that department is named EmployeesInDepartment. The application module will have a DepartmentsTree view object instance of type DeptVO. It will also have an EditDepartment view object instance of type DeptVO, and a view link for the EditEmployees view object instance of type EmpVO.

1. Drag the DepartmentsTree data collection from the data control palette onto the page and choose **Create -> Trees -> ADF Tree**.
2. Configure tree binding rules to navigate the EmployeesInDepartment view link accessor attribute.

This will access the children employee rows of the current department row.

3. Drag the EditEmployees detail view object instance to the page, and choose **Create -> Master-Detail -> ADF Master Form, Detail Form**.

This creates a form to edit a department and a form to edit an employee in that department.

4. For the DeptVO node definition of the tree binding, configure the TargetIterator property to be `#{bindings.EditDepartmentIterator}`.
5. For the EmpVO node definition of the tree binding, configure the TargetIterator property to be `#{bindings.EditEmployeesIterator}`.
6. When you run the master-detail form, clicking on an employee in any department in the tree will first set the current department row in the target iterator for the department of that selected employee. Then it will set the current employee row in the target iterator for the selected employee.

23

Creating Databound Selection Lists and Shuttles

The chapter describes how to add selection lists and shuttle components to pages. This chapter contains the following sections:

- [Chapter 23.1, "Creating a Selection List"](#)
- [Chapter 23.2, "Creating a List with Navigation List Binding"](#)
- [Chapter 23.3, "Creating a Databound Shuttle"](#)

23.1 Creating a Selection List

ADF Faces Core includes components for selecting a single and multiple values from a list. Single selection lists are described in [Table 23–1](#).

Table 23–1 ADF Faces Single and Multiple List Components

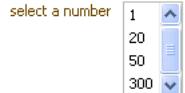
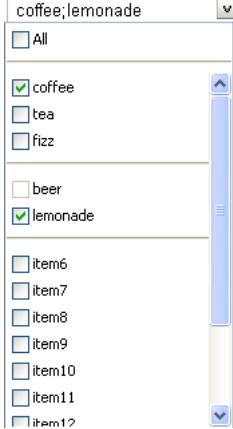
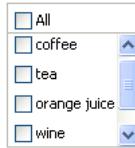
ADF Faces component	Description	Example
SelectOneChoice	Select a single value from a list of items	
SelectOneRadio	Select a single value from a set of radio buttons	
SelectOneListbox	Select a single value from a scrollable list of items	

Table 23–1 (Cont.) ADF Faces Single and Multiple List Components

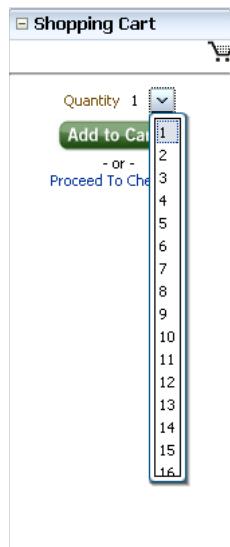
ADF Faces component	Description	Example
SelectManyChoice	Select a multiple values from a scrollable list of check boxes. Each selection displays at the top of the list.	
SelectManyCheckbox	Select multiple values from a group of check boxes	
SelectManyListbox	Select multiple values from a scrollable list of check boxes	

These components work in the same way as standard JSF list components. ADF Faces list components, however, provide extra functionality such as support for label and message display, automatic form submission, and partial page rendering.

List of Values (LOV) are UI components that allow the user to enter values by picking from a list that is generated by a query. The LOV displays inside a pop up modal dialog that typically includes search capabilities. The `af:inputListOfValues` and `af:inputComboboxListOfValues` components, for example, offer additional features that are not available in selection lists such as search fields inside the LOV modal dialog and queries based on multiple table columns. For more information, see [Section G.2.6, "How to Create a Popup List of Values"](#) and the *Oracle Fusion Middleware Web User Interface Developer's Guide for Oracle Application Development Framework*.

23.1.1 How to Create a Single Selection List

This section describes how to create selection lists using the `SelectOneChoice` ADF Faces component, but the steps are similar for creating other single value selection lists, such as `SelectOneRadio` and `SelectOneListbox`. As shown [Figure 23–1](#), the Fusion Order Demo Home page uses a `SelectOneChoice` component that allows a user to choose item quantities when adding a selected item to the Shopping Cart region.

Figure 23–1 SelectOneChoice List Component

The `SelectOneChoice` component is located on the `embeddedCartSummary.jsff` page fragment. The page fragment is a view activity on the `embeddedCartSummary-task-flow` bounded task flow, which is added to the Home page as an ADF region.

A databound selection list displays values from a data control collection or a static list and updates an attribute in another collection or a method parameter based on the user's selection. When adding a binding to a list, you use an attribute from the data control that will be populated by the selected value in the list.

To create a selection list, you choose a base data source and list data source:

- **Base data source:** Contains the data that will be updated when the end user selects a value in the list. For example, selecting a number in the `SelectOneChoice` list in the Shopping Cart updates the `OrderItems` data collection.
- **List data source:** Contains the data used to populate the list. For example, the `quantities` attribute of the `ProductQuantities` view supplies the values that display in the list shown in [Figure 23–1](#).

`ProductQuantities` is a view collection object that contains the `Quantity`, `QuantityOnHand`, and `Product_Id` attributes. The view collection object was created based on attributes in the `WarehouseStockLevels` and `Orders` data collections. For more information about creating a view object, see [Section 5.2.1, "How to Create an Entity-Based View Object"](#).

You can create three types of selection lists in the List Binding Editor:

- **Model-driven list** (recommended): List selections are based on a List of Values bound to a data collection. This type of selection list offers significant advantages over the other two, as described in [Section 23.1.2, "How to Create a Model-Driven List"](#).
- **Static list:** List selections are based on a fixed list that you create manually by entering values one at a time into the editor. See [Section 23.1.3, "How to Create a Selection List Containing Fixed Values"](#) for more information.

- **Dynamic list:** List selections are generated dynamically based on one or more databound attribute values. See [Section 23.1.4, "How to Create a Selection List Containing Dynamically Generated Values"](#) for more information.

23.1.2 How to Create a Model-Driven List

Note: The recommended way to create a model-driven list is to drag a collection from the data control palette onto a JSF page, choose one of the ADF Forms in the pop up menu, and accept the defaults. The advantage of this is that if there are LOVs defined on the underlying view object attributes, all the LOVs on the entire form will be configured automatically. For more information, see [Section 5.11.1, "How to Define a Single LOV-Enabled View Object Attribute"](#)

The section below describes dragging individual attributes onto a JSF page and choosing **Model Driven List**. This is also a valid approach for creating a model-driven list, but you should first consider using the recommended method described above.

A model-driven list is based on a List of Values that is bound to a view data object. Lists of Values are typically used in forms to enable an end user to select an attribute value from a drop down list instead of having to enter it manually. When the user submits the form with the selected values, ADF data bindings in the ADF Model layer update the value on the view object attributes corresponding to the databound fields.

You can also use the List of Values as the basis for creating a selection list. The advantages of creating a model-driven list based on a List of Values are:

- **Reuse:** The List of Values is bound to a view data collection. Any selection list that you create based on the data collection can use the same List of Values. Because you define the LOV on the individual attributes of business components, you can customize the LOV usage for an attribute once and expect to see the changes anywhere the business component is used in the user interface.
- **Translation:** Values in the List of Values can be included in resource bundles used for translation.

Before you can create the selection list, you must create a List of Values that is bound to an attribute on the base data source for the selection list. For example, you can create a List of Values bound to the `CountryId` attribute of the `Addresses` view data object. The procedure for creating a List of Values is simple:

1. Create a view object.
2. Create a view accessor on the object.
3. Create a List of Values on an attribute of the view object.

This procedure is described in detail in [Section 5.11.1, "How to Define a Single LOV-Enabled View Object Attribute"](#).

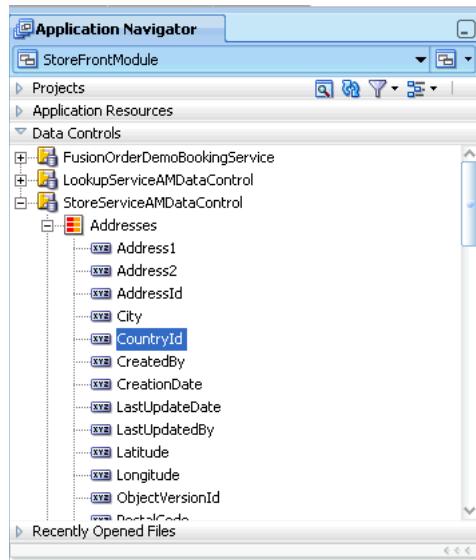
To create a model-driven selection list:

1. In the editor, open the JSF page where you plan to add the selection list.
2. In the Data Control panel, expand the view object collection that will be the base data source.

This is the data collection contains the attribute on which you created the List of Values according to the steps in [Section 5.11.1, "How to Define a Single LOV-Enabled View Object Attribute"](#). In the Fusion Order Demo application, for example, you could select the Addresses view object collection as the base data source.

3. In the Data Control panel, select the attribute under the view object collection, for example, CountryId.

Figure 23–2 View Attribute in StoreFrontModule



4. Drag the attribute from the Data Control panel and drop it on the JSF page.
5. Choose **Create > Single Selections > ADF Select One Choice**.

The List Binding Editor displays. The view object collection containing the attribute you dropped on the JSF page is selected by default in the **Base Data Source** list.

To select a different view data collection, click the **Add** button next to the list.

6. Select the **Model Driven List** radio button.
7. In the **Base Data Source Attribute** list, select an attribute on which you based a List of Values, for example, CountryId.

The list contains all the attributes for the view data collection selected in the Base Data Source list. For example, if you selected Addresses as the base data source, you could select as its CountryId attribute.

8. If a List of Values was created for the attribute you selected, it will be listed in the **Server List Binding Name** list.

For example, you could select LOV_CountryId in the **Server List Binding Name** list because a List of Values was created for the CountryId attribute.

9. Click **OK**.

23.1.3 How to Create a Selection List Containing Fixed Values

You can create a selection list containing selections that you code yourself, rather than getting the values from another data source (see [Section 23.1.4, "How to Create a Selection List Containing Dynamically Generated Values"](#) for information about populating selection list with values that are dynamically generated from another data source.

Figure 23–3 Selection List Bound to a Fixed List of Values



To create a list bound to a fixed list of values:

1. In the editor, open the JSF page where you plan to add the selection list.
2. In the Data Control panel, expand the view object collection that will be the base data source.

This data is updated when the end user selects a value in the selection list. In the Fusion Order Demo application, for example, you could select the `CountryCodes` view object collection as the base data source.

3. In the Data Control panel, select an attribute under the view object collection, for example, `CountryName`.
4. Drag the attribute from the Data Control panel and drop it on the JSF page.
5. Choose **Create > Single Selections > ADF Select One Choice**.

The List Binding Editor displays. The view object collection containing the attribute you dropped on the JSF page is selected by default in the **Base Data Source** list.

To select a different view data collection, click the **Add** button next to the list.

6. Select the **Fixed List** radio button.

The **Fixed List** option lets end users choose a value from a static list that you define.

7. In the **Base Data Source Attribute** torpedoing list, choose an attribute.

The **Base Data Source Attribute** list contains all of the attributes in the view data collection you selected in the **Base Data Source** list. For example, if you selected `CountryCodes` as the **Base Data Source**, you can choose `CountryName` in the list.

8. In the **Set of Values** box, enter each value you want to appear in the list. Press the `Enter` key to set a value before typing the next value. For example, you could add the following list of country codes

- India
- Japan
- Russia

The order in which you enter the values is the order in which the list items are displayed in the `SelectOneRadio` control at runtime.

The `SelectOneRadio` component supports a `null` value. If the user has not selected an item, the label of the item is shown as blank, and the value of the component defaults to an empty string. Instead of using blank or an empty string, you can specify a string to represent the `null` value. By default, the new string appears at the top of the list.

9. Click **OK**.

23.1.4 How to Create a Selection List Containing Dynamically Generated Values

You can populate a selection list component with values dynamically at runtime. The steps for creating a list component bound to a dynamic list are almost the same as those for creating a list component bound to a fixed list, with the exception that you define two data sources: one for the list data source that provides the dynamic list of values, and the other for the base data source that is to be updated based on the user's selection.

To create a selection list bound containing dynamically-generated values:

1. In the editor, open the JSF page where you plan to add the selection list.
2. In the Data Control panel, expand the view object collection that will be the base data source.
This data is updated when the end user selects a value in the selection list. In the Fusion Order Demo application, for example, you could select the `OrderItems` view object collection as the base data source.
3. In the Data Control panel, select an attribute under the view object collection, for example, `Quantities`.
4. Drag the attribute from the Data Control panel and drop it on the JSF page.
5. Choose **Create > Single Selections > ADF Select One Choice**.

The List Binding Editor displays. The view object collection containing the attribute you dropped on the JSF page is selected by default in the **Base Data Source** list.

- To select a different view data collection, click the **Add** button next to the list.
6. Select the **Dynamic List** radio button.
The Dynamic List option lets you specify one or more base data source attributes that will be updated from another set of bound values.
 7. Click the **Add** button next to **List Data Source**.
 8. In the Add Data Source dialog, select the view data collection that will populate the values in the selection list.
In the Fusion Order Demo application, for example, you could select `ProductQuantities`.

Note: The list and base collections do not have to form a master-detail relationship, but the attribute in the list collection must have the same type as the base collection attributes.

9. Accept the default **Iterator Name** and click **OK**.

The **Data Mapping** section of the Edit List Binding dialog updates with a default **Data Value** and **List Attribute**. **Data Value** contains the attribute on the data

collection that is updated when the user selects a item in the selection list. List Attribute contains the attribute that populates the values in the selection list.

10. You can accept the default mapping or select different attributes items from the **Data Value** and **List Attribute** lists to update the mapping.

To add a second mapping, click **Add**.

11. Click **OK**.

23.1.5 What Happens When You Create a Model-driven Selection List

When you drag and drop an attribute from the Data Control panel, JDeveloper does many things for you. For a full description of what happens and what is created when you use the Data Control panel, see [Section 20.2.2, "What Happens When You Create a Text Field"](#).

[Example 23–1](#) shows the page source code for after you add a model-driven `SelectOneChoice` component to it.

Example 23–1 Model-driven SelectOneChoice List in JSF Page Source Code

```
<af:selectOneChoice value="#{bindings.CountryId1.inputValue}"  
                    label="#{bindings.CountryId1.label}">  
    <f:selectItems value="#{bindings.CountryId1.items}" />  
</af:selectOneChoice>
```

The `f:selectItems` tag, which provides the list of items for selection, is bound to the `items` property on the `CountryId1` list binding object in the binding container. For more information about ADF data binding expressions, see [Section 11.7, "Creating ADF Data Binding EL Expressions"](#).

In the page definition file, JDeveloper adds the list binding object definitions in the `bindings` element, as shown in [Example 23–2](#).

Example 23–2 List Binding Object for the Model-Driven List in the Page Definition File

```
<bindings>  
    <list IterBinding="AddressesView1Iterator" id="CountryId"  
          Uses="LOV_AddressId" StaticList="false" ListOperMode="0">  
        <AttrNames>  
            <Item Value="AddressId"/>  
        </AttrNames>  
    </list>  
    <list IterBinding="AddressesView1Iterator" id="CountryId1"  
          Uses="LOV_CountryId" StaticList="false" ListOperMode="0">  
        <AttrNames>  
            <Item Value="CountryId"/>  
        </AttrNames>  
    </list>  
</bindings>
```

In the `list` element, the `id` attribute specifies the name of the list binding object. The `IterBinding` attribute references the variable iterator, whose current row is a row of attributes representing each variable in the binding container. The variable iterator exposes the variable values to the bindings in the same way as other collections of data. The `AttrNames` element specifies the attribute value returned by the iterator.

For more information about the page definition file and ADF data binding expressions, see [Section 11.6, "Working with Page Definition Files"](#) and [Section 11.7, "Creating ADF Data Binding EL Expressions"](#).

23.1.6 What Happens When You Create a Fixed Selection List

[Example 23–3](#) shows the page source code after you add a fixed SelectOneChoice component to it.

Example 23–3 Fixed SelectOneChoice List in JSF Page Source Code

```
<af:selectOneChoice value="#{bindings.CountryId.inputValue}"
    label="#{bindings.CountryId.label}">
    <f:selectItems value="#{bindings.CountryId.items}" />
</af:selectOneChoice>
```

The f:selectItems tag, which provides the list of items for selection, is bound to the items property on the CountryId list binding object in the binding container. For more information about ADF data binding expressions, see [Section 11.7, "Creating ADF Data Binding EL Expressions"](#).

In the page definition file, JDeveloper adds the definitions for the iterator binding objects into the executables element, and the list binding object into the bindings element, as shown in Example 19–5.

Example 23–4 List Binding Object for the Fixed Selection List in the Page Definition File

```
<executables>
    <iterator Binds="Addresses1" RangeSize="10"
        DataControl="StoreFrontModuleDataControl"
        id="Addresses1Iterator"/>
</executables>
<bindings>
    <list IterBinding="Addresses1Iterator" id="CountryId" ListOperMode="0"
        StaticList="true">
        <AttrNames>
            <Item Value="CountryId"/>
        </AttrNames>
        <ValueList>
            <Item Value="India"/>
            <Item Value="Japan"/>
            <Item Value="Russia"/>
        </ValueList>
    </list>
</bindings>
```

For complete information about page definition files, see [Section 11.6, "Working with Page Definition Files"](#).

23.1.7 What You May Need to Know About Values in a Selection List

Once you have created a list binding, you may want to access a value in the list. If you attempt to get the value of the list binding directly using an EL expression, for example, #{bindings.deptList.inputValue}, the expression returns an index number that specifies the position of the selected item in the list, not the value of the selected item.

23.1.8 What Happens When You Create a Dynamic Selection List

[Example 23–5](#) shows the page source code after you add a dynamic SelectOneChoice component to it.

Example 23–5 Dynamic SelectOneChoice List in JSF Page Source Code

```
<af:selectOneChoice value="#{bindings.Quantity.inputValue}"  
                    label="#{bindings.Quantity.label}">  
    <f:selectItems value="#{bindings.Quantity.items}" />  
</af:selectOneChoice>
```

The `f:selectItems` tag, which provides the list of items for selection, is bound to the `items` property on the `Quantity` list binding object in the binding container. For more information about ADF data binding expressions, see [Section 11.7, "Creating ADF Data Binding EL Expressions"](#).

In the page definition file, JDeveloper adds the definitions for the iterator binding objects into the `executables` element, and the list binding object into the `bindings` element, as shown in [Figure 23–6](#).

Example 23–6 List Binding Object for the Dynamic Selection List in the Page Definition File

```
<executables>  
    <iterator Binds="OrderItems" RangeSize="-1"  
              DataControl="StoreFrontModuleDataControl1"  
              id="OrderItemsIterator"/>  
    <iterator Binds="ProductQuantities" RangeSize="10"  
              DataControl="StoreFrontModuleDataControl1"  
              id="ProductQuantitiesIterator"/>  
</executables>  
<bindings>  
    <list IterBinding="AddressesView1Iterator" id="CountryId"  
          Uses="LOV_AddressId" StaticList="false" ListOperMode="0">  
        <AttrNames>  
            <Item Value="AddressId"/>  
        </AttrNames>  
    </list>  
    <list IterBinding="ProductQuantities1Iterator" id="Quantity"  
          StaticList="false" ListOperMode="0" ListIter="OrderItems1Iterator">  
        <AttrNames>  
            <Item Value="Quantity"/>  
        </AttrNames>  
        <ListAttrNames>  
            <Item Value="Quantity"/>  
        </ListAttrNames>  
        <ListDisplayAttrNames>  
            <Item Value="OrderId"/>  
        </ListDisplayAttrNames>  
    </list>  
</bindings>
```

By default, JDeveloper sets the `RangeSize` attribute on the `iterator` element for the `OrderItems` iterator binding to a value of `-1`, thus allowing the iterator to furnish the full list of valid products for selection. In the `list` element, the `id` attribute specifies the name of the list binding object. The `IterBinding` attribute references the iterator that iterates over the `ProductQuantities` collection. The `ListIter` attribute references the iterator that iterates over the `ProductList` collection. The `AttrNames` element specifies the base data source attributes returned by the base iterator. The `ListAttrNames` element defines the list data source attributes that are mapped to the base data source attributes. The `ListDisplayAttrNames` element specifies the list data source attribute that populates the values users see in the list at runtime.

For complete information about page definition files, see [Section 11.6, "Working with Page Definition Files"](#).

23.2 Creating a List with Navigation List Binding

Navigation list binding lets users navigate through the objects in a collection. As the user changes the current object selection using the navigation list component, any other component that is also bound to the same collection through its attributes will display from the newly selected object. In addition, if the collection whose current row you change is the master view object instance in a data model master/detail relationship, the row set in the detail view object instance is automatically updated to show the appropriate data for the new current master row.

To create a list that uses navigation list binding:

1. In Data Control panel, expand a project node, for example, `StoreFrontAMDataControl`.
2. Drag and drop the a collection to the page, for example Addresses, and choose **Create > Navigation > ADF Navigation Lists**. The List Binding Editor displays. For example, you might drop **StaffList** because you want users to navigate and select a technician name.
3. In the **Display Attributes** section, double-click the attributes of the `StaffList` collection that you don't want to be displayed in the list, moving them to the **Available Attributes** section.
4. In the **Select an Iterator** dropdown list, the default iterator name for the collection is shown. Accept the default or click **New** to create a new name for the iterator.

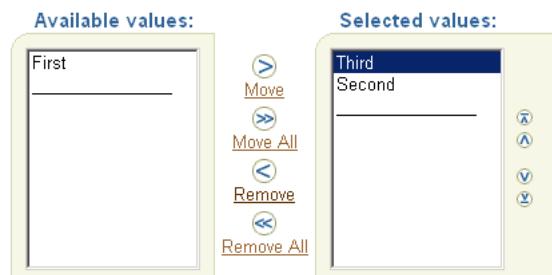
23.3 Creating a Databound Shuttle

The `selectManyShuttle` and `selectOrderShuttle` components render two list boxes, and buttons that allow the user to select multiple items from the leading (or available) list box and move or shuttle the items over to the trailing (or selected) list box, and vice versa. [Figure 23–4](#) shows an example of a rendered `selectManyShuttle` component. You can specify any text you want for the headers that display above the list boxes.

Figure 23–4 Shuttle (SelectManyShuttle) Component



The only difference between `selectManyShuttle` and `selectOrderShuttle` is that in the `selectOrderShuttle` component, the user can reorder the items in the trailing list box by using the up and down arrow buttons on the side, as shown in [Figure 23–5](#).

Figure 23–5 Shuttle Component (*SelectOrderShuttle*) with Reorder Buttons

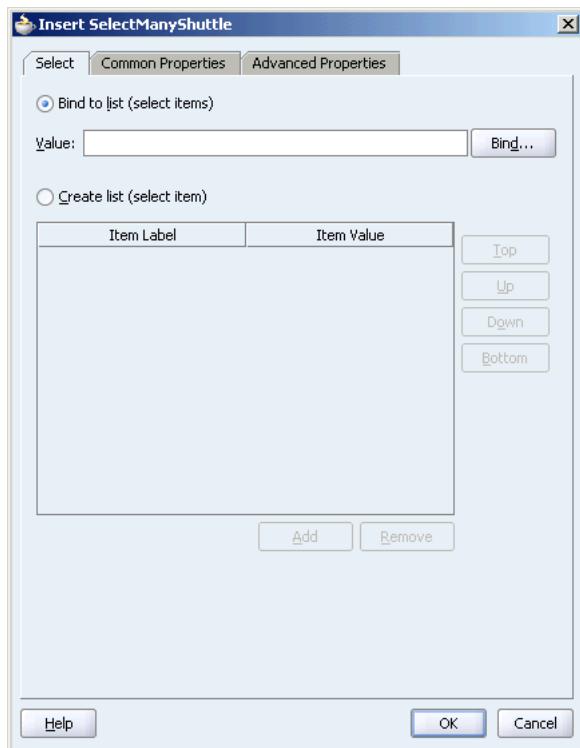
The Fusion Order Demo uses a `selectManyShuttle` component to move products from a Selected list box to a Purchase list box. The leading list box on the left displays products. The trailing list box on the right displays the products that the customer has purchased.

23.3.1 How to Create a Shuttle

Like other ADF Faces selection list components, the `selectManyShuttle` component can use the `f:selectItems` tag to provide the list of items available for display and selection in the leading list.

To create a shuttle component that is associated with a navigation list component:

1. In the JSF page that contains the navigation list component, select the `selectOneChoice` component.
2. In the Property Inspector, set the **Id** attribute to a value (for example, `AddressId`).
3. Set the **ValueChangeListener** attribute to the `refreshSelectedList()` method in a managed bean (for example, `#{backingFusionDemo.refreshSelectedList}`).
4. On the **ADF Faces Core** page of the Component Palette, drag and drop **SelectManyShuttle** to the page. JDeveloper displays the Insert **SelectManyShuttle** dialog, as illustrated in [Figure 23–6](#).

Figure 23–6 Insert SelectManyShuttle Dialog

5. Select **Bind to list (select items)** and click **Bind...** to open the Expression Builder.
6. In the Expression Builder, expand **JSF Managed Beans > backing_SRSkills**. Double-click **allItems** to build the expression `# {backing_SRSkills.allItems}`. Click **OK**.

This binds the `f:selectItems` tag to the `getAllItems()` method that populates the shuttle's leading list.

7. In the Insert SelectManyShuttle dialog, click **Common Properties**. Click **Bind...** next to the **Value** field to open the Expression Builder again.
8. In the Expression Builder, expand **JSF Managed Beans > backing_SRSkills**. Double-click **selectedValues** to build the expression `# {backingFusionDemo.refreshSelectedList}`.

This binds the `value` attribute of the `selectManyShuttle` component to the `getSelectedValues()` method that populates the shuttle's trailing list.

9. Click **OK**.
10. Close the Insert SelectManyShuttle dialog.
11. In the Property Inspector, set the `partialTriggers` attribute on the `selectManyShuttle` component to the Id of the `selectOneChoice` component (for example, `technicianList`) that provides the navigation dropdown list of technician names.

[Example 23–7](#) shows the code for the `selectManyShuttle` component after you complete the Insert SelectManyShuttle dialog.

Example 23–7 SelectManyShuttle Component in the SRSkills.jspx File

```
<af:selectManyShuttle value="#{backing_SRSkills.selectedValues}"
```

```
partialTriggers="technicianList">
...
<f:selectItems value="#{backing_SRSkills.allItems}"/>
</af:selectManyShuttle>
```

Creating Databound ADF Data Visualization Components

This chapter describes how to use the Data Controls panel to create ADF Data Visualization components. These components allow you to display and analyze data through a wide variety of graphs, several kinds of gauges, a pivot table, geographic maps with multiple layers of information, and several kinds of gantt.

This chapter includes the following sections:

- [Section 24.1, "Introduction to Creating ADF Data Visualization Components"](#)
- [Section 24.2, "Creating Databound Graphs"](#)
- [Section 24.3, "Creating Databound Gauges"](#)
- [Section 24.4, "Creating Databound Pivot Tables"](#)
- [Section 24.5, "Creating Databound Geographic Maps"](#)
- [Section 24.6, "Creating Databound Gantt Charts"](#)

24.1 Introduction to Creating ADF Data Visualization Components

ADF Data Visualization components provide extensive graphical and tabular capabilities for visually displaying and analyzing data.

Both ADF graphs and ADF gauges render graphical representations of data. However, graphs (which produce more than 50 types of charts) allow you to evaluate multiple data points on multiple axes in a variety of ways. Many graph types assist in the comparison of results from one group against the results from another group. In contrast, gauges focus on a single data point and examine that point relative to minimum, maximum, and threshold indicators to identify problems.

The ADF pivot table produces a grid that supports multiple layers of data labels on the row edge or the column edge of the grid. This component also provides the option of automatically generating subtotals and totals for grid data. ADF pivot tables let you switch data labels from one edge to another to obtain different views of your data. For example, a pivot table might initially display products within region in its rows while showing years in its columns. If you switch region to the column edge so that columns display year within region, then data cells in the table show totals for products by year within region. At runtime, end users can click buttons that appear in the inner column labels to sort rows in ascending or descending order. If an end user clicks one of these buttons, it changes the default behavior where changes to the outer row labels sort rows.

The ADF geographic map is a component that represents business data and allows you to superimpose multiple layers (also referred to as *themes*) of information on a single map. For example, a map of the United States might use a color theme that provides varying color intensity to indicate the popularity of a product within each state, a pie chart theme that shows sales within product category, and a point theme that identifies the exact location of each warehouse. When all three themes are superimposed on the United States map, you can easily evaluate whether there is sufficient inventory to support the popularity level of a product in specific locations.

There are three types of ADF gantt component; these are the project gantt (which focuses on project management), the scheduling gantt, and the resource utilization gantt (which both focus on resource management). Each gantt shows the following regions combined with a splitter:

- List region content: The left side of the splitter provides a list of tasks (for the project gantt) and a list of resources (for the resource utilization and scheduling gantts). This region can display any number of additional columns of related information.
- Chart region content: The right side of the splitter consists of an area in which task progress, resource utilization, or resource progress is graphed over time. The ability of the gantt to zoom in or out on its time axis lets you view management information across the desired time period.

Each ADF Data Visualization component needs to be bound to data before it can be rendered because the appearance of the components is dictated by the data that is displayed. This chapter describes how to bind each component to a data source.

[Figure 24-1](#) shows a number of the ADF Data Visualization components at runtime including a bar graph, a pie chart graph, a dial gauge, and a status meter gauge.

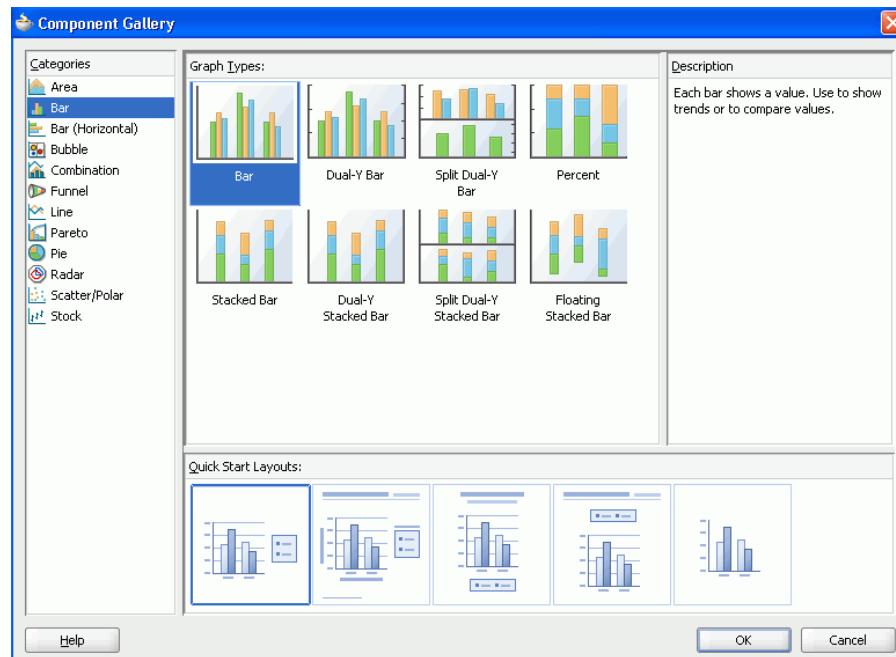
Figure 24–1 Dashboard with ADF Data Visualization Components

24.2 Creating Databound Graphs

When you create a graph, a Component Gallery allows you to choose from a wide number of graph categories, graph types, and layout options. Graph categories group together one or more different types of graph. For example, the Scatter/Polar category includes the following types of graph:

- Scatter
- Dual-Y Scatter
- Polar

Explore the Component Gallery that appears when you create a graph to view all available graph categories, types, and descriptions for each one. [Figure 24–2](#) shows the Component Gallery that appears for ADF graphs.

Figure 24–2 ADF Graphs Component Gallery

Note: Some graph types require very specific kinds of data. If you bind a graph to a data collection that does not contain sufficient data to display the graph type requested, then the graph is not displayed and a message about insufficient data appears.

Table 24–1 lists the categories that appear in the Component Gallery for ADF graphs. Each category has one or more graph types associated with it.

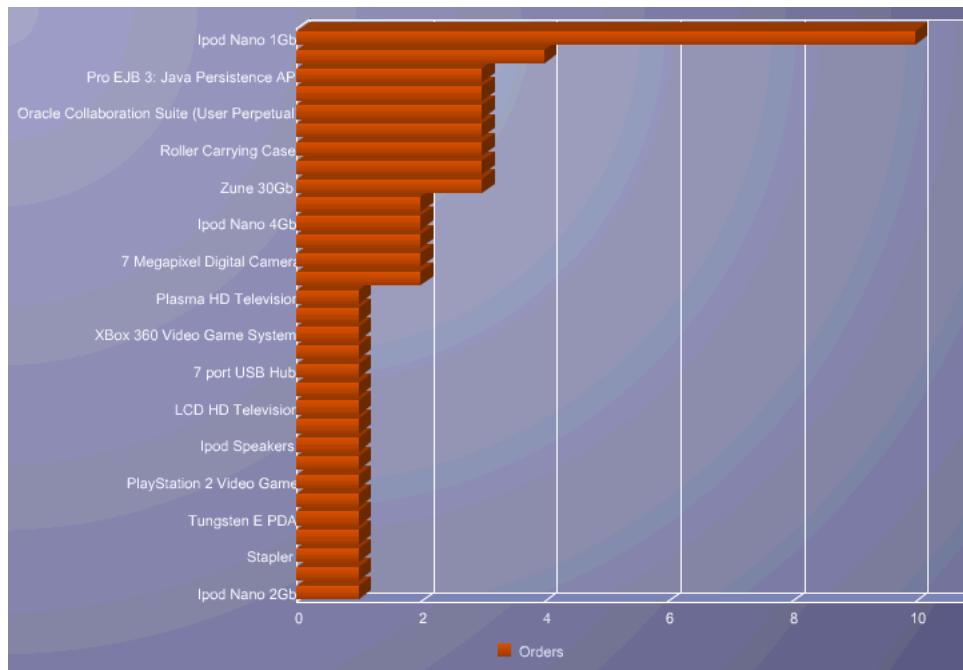
Table 24–1 ADF Graph Categories in the Component Gallery

Category	Description
Area	Creates a graph in which data is represented as a filled-in area. Use area graphs to show trends over time, such as sales for the last 12 months. Area graphs require at least two groups of data along an axis. The axis is often labeled with time periods such as months.
Bar	Creates a graph in which data is represented as a series of vertical bars. Use to examine trends over time or to compare items at the same time, such as sales for different product divisions in several regions.
Bar (Horizontal)	Creates a graph that displays bars horizontally along the y-axis. Use to provide an orientation that allows you to show trends or compare values.
Bubble	Creates a graph in which data is represented by the location and size of round data markers (bubbles). Use to show correlations among three types of values, especially when you have a number of data items and you want to see the general relationships. For example, use a bubble graph to plot salaries (x-axis), years of experience (y-axis), and productivity (size of bubble) for your work force. Such a graph allows you to examine productivity relative to salary and experience.
Combination	Creates a graph that uses different types of data markers (bars, lines, or areas) to display different kinds of data items. Use to compare bars and lines, bars and areas, lines and areas, or all three.

Table 24–1 (Cont.) ADF Graph Categories in the Component Gallery

Category	Description
Funnel	Creates a graph that is a visual representation of data related to steps in a process. The steps appear as vertical slices across a horizontal cylinder. As the actual value for a given step or slice approaches the quota for that slice, the slice fills. Typically a funnel graph requires actual values and target values against a stage value, which might be time. For example, use this component to watch a process (such as a sales pipe line) move towards a target across the stage of the quarters of a fiscal year.
Line	Creates a graph in which data is represented as a line, as a series of data points, or as data points that are connected by a line. Line graphs require data for at least two points for each member in a group. For example, a line graph over months requires at least two months. Typically a line of a specific color is associated with each group of data such as the Americas, Europe, or Asia. Use to compare items over the same time.
Pareto	Creates a graph in which data is represented by bars and a percentage line that indicates the cumulative percentage of bars. Each set of bars identifies different sources of defects, such as the cause of a traffic fatality. The bars are arranged by value, from the largest number to the lowest number of incidents. A pareto graph is always a dual-y graph in which the first y-axis corresponds to values that the bars represent and the second y-axis runs from 0 to 100% and corresponds to the cumulative percentage values. Use the pareto graph to identify and compare the sources of defects.
Pie	Creates a graph in which one group of data is represented as sections of a circle causing the circle to look like a sliced pie. Use to show relationship of parts to a whole such as how much revenue comes from each product line.
Radar	Creates a graph that appears as a circular line graph. Use to show patterns that occur in cycles, such as monthly sales for the last three years.
Scatter/Polar	Creates a graph in which data is represented by the location of data markers. Use to show correlation between two different kinds of data values such as sales and costs for top products. Scatter graphs are especially useful when you want to see general relationships among a number of items.
Stock	Creates a graph in which data shows the high, low, and closing prices of a stock. Each stock marker displays three separate values.

Figure 24–3 shows the bar graph that appears in the Hot Items Statistics page of the StoreFrontModule application. This graph displays the quantity of orders for each product so that you can easily identify the items that have been ordered most frequently.

Figure 24–3 Hot Items Statistics Graph

For information about customizing graphs after the data-binding is completed, see the "Using ADF Graph Components" chapter in *Oracle Fusion Middleware Web User Interface Developer's Guide for Oracle Application Development Framework*.

24.2.1 How to Create a Graph

Graphs are based on data collections. Using data controls in Oracle ADF, Oracle JDeveloper makes this a declarative task. You drag and drop a collection from the Data Controls panel.

To create a databound graph:

- From the Data Controls panel, select a collection.

For example, to create the bar graph that displays the order count for each product in the Hot Items Statistics page of the StoreFrontModule application, you would select the `ProductOrdersCount` collection. Figure 24–4 shows the `ProductOrdersCount` collection in the Data Controls panel.

Figure 24–4 Data Collection for Counting Product Orders

- Drag the collection onto a JSF page and, from the context menu, choose **Graphs** to display the Component Gallery.

3. Select a graph category and a graph type from the Component Gallery and click **OK**.

For information about the graph categories and graph types that appear in the Component Gallery, see [Table 24–1, " ADF Graph Categories in the Component Gallery"](#).

The name of the dialog that appears depends on the category and type of graph that you select. For example, if you select Bar (Horizontal) as the category and Bar as the type, then the name of the dialog that appears is Create Horizontal Bar Graph.

4. Do the following in the dialog to configure the graph to display data:

- Drag attributes from the **Available** list to the **Areas** or **X Axis** input fields, depending on where you want the values for the attributes to appear at runtime.
- In the **Attribute Labels** field, enter a value or select a value from the dropdown list in the **Label** column to specify the label that appears at runtime.
- If you want to change from the default selection of **Typed Attributes** to **Name-Value Pairs** to configure how data points are stored in a collection, then click the **Change Data Shape** button. A dialog appears that presents you with the following options:

- **Typed Attributes**

Each kind of data point in the collection is represented by a different attribute. This option is also valid when there is only a single kind of data point in the graph.

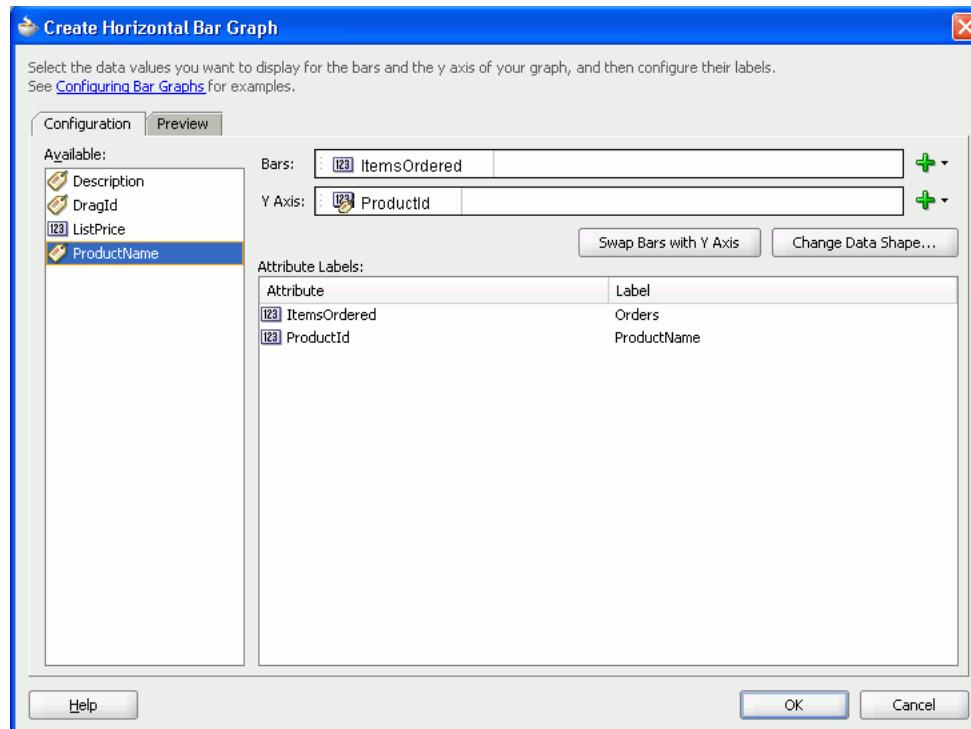
For example, if you have data points for **Estimated Value** and **Actual Value**, then select **Typed Attributes** only if you have one attribute for the estimated value and a second attribute for the actual value.

- **Name-Value Pairs**

Indicates that there are multiple kinds of data points, but only a single attribute to designate these points. In this case, the single attribute has values that identify each kind of data point.

For example, the attribute might have the value EST for a data point that represents an estimated value and ACT for a data point that represents an actual value.

[Figure 24–5](#) shows the Create Horizontal Bar Graph that generates a graph using data from the `ItemsOrdered` attribute in the `ProductOrdersCount` data collection. The `ProductName` attribute provides labels for the displayed data.

Figure 24–5 Create Horizontal Bar Graph Dialog for ProductOrdersCount

5. Optionally, click the **Preview** tab to do the following:
 - Display a live preview of the bar graph and its data
 - As needed, click the **Configuration** tab so that you can adjust the bar graph specifications
6. Click **OK**.

After completing the data binding dialog, you can use the Property Inspector to specify settings for the graph attributes and you can also use the child tags associated with the graph tag to customize the graph further.

24.2.2 What Happens When You Use the Data Controls Panel to Create a Graph

Dropping a graph from the Data Controls panel has the following effect:

- Creates the bindings for the graph and adds the bindings to the page definition file
- Adds the necessary code for the UI components to the JSF page

The data binding XML that JDeveloper generates varies depending on the type of graph you select. The XML represents the physical model of the specific graph type you create. [Example 24–1](#) shows the bindings that JDeveloper generated in the page definition file where a horizontal bar graph was created using data from the `ItemsOrdered` attribute in the `ProductOrdersCount` data collection.

Example 24–1 Binding XML for an ADF Bar (Horizontal) Graph

```
<graph IterBinding="ProductOrdersCountIterator" id="ProductOrdersCount"
       xmlns="http://xmlns.oracle.com/adfm/dvt" type="BAR_HORIZ_CLUST">
  <graphDataMap leafOnly="true">
    <series>
      <data>
```

```

        <item value="ItemsOrdered"/>
    </data>
</series>
<groups>
    <item value="ProductName"/>
</groups>
</graphDataMap>
</graph>

```

Example 24–2 shows the code generated for a horizontal bar graph when you drag the ProductOrdersCount data collection onto a JSF page.

Example 24–2 JSF Code for an ADF Bar (Horizontal) Graph

```

<dvt:horizontalBarGraph id="horizontalBarGraph1"
    value="#{bindings.ProductOrdersCount.graphModel}"
    subType="BAR_HORIZ_CLUST">
    <dvt:background>
        <dvt:specialEffects/>
    </dvt:background>
    <dvt:graphPlotArea/>
    <dvt:seriesSet>
        <dvt:series/>
    </dvt:seriesSet>
    <dvt:o1Axis/>
    <dvt:y1Axis/>
    <dvt:legendArea automaticPlacement="AP_NEVER" />
</dvt:horizontalBarGraph>

```

24.2.3 What You May Need to Know About Using a Graph's Row Selection Listener for Master-Detail Processing

You can use the row selection listener of a graph (which serves as a master view) to enable clicks on a bar, slice, or other graph data element to update the data in another ADF component (which serves as a detail view). For example, a click on a bar that represents sales for a given product in a graph might cause the display of the detailed sales data related to the product in a pivot table.

The following requirements must be met to achieve this master-detail processing declaratively:

1. You must use the same tree data control to provide data for both views as follows:
 - a. Bind the graph as a row set to one level in the data control.
 - b. Bind the other ADF view (such as a table or pivot table) to a lower level in the tree data control.
2. Set a value for the `clickListener` attribute of the graph tag in the Behavior page of the Property Inspector and use the `processClick` method that is already part of the graph binding.

For example, if the `value` attribute of the graph tag is
`value="#{bindings.myGraph.graphModel}"`, then the `clickListener` attribute should be
`clickListener="#{bindings.myGraph.graphModel.processClick}"`.

3. Ensure that the `partialTriggers` attribute on the parent tag for the detail component is set correctly. It should be set to the ID of the graph component.

You do not have to write Java code for handling clicks on data elements in the graph. The `processClick` event on the graph binding recognizes click events on data component in a graph and performs the necessary processing.

24.3 Creating Databound Gauges

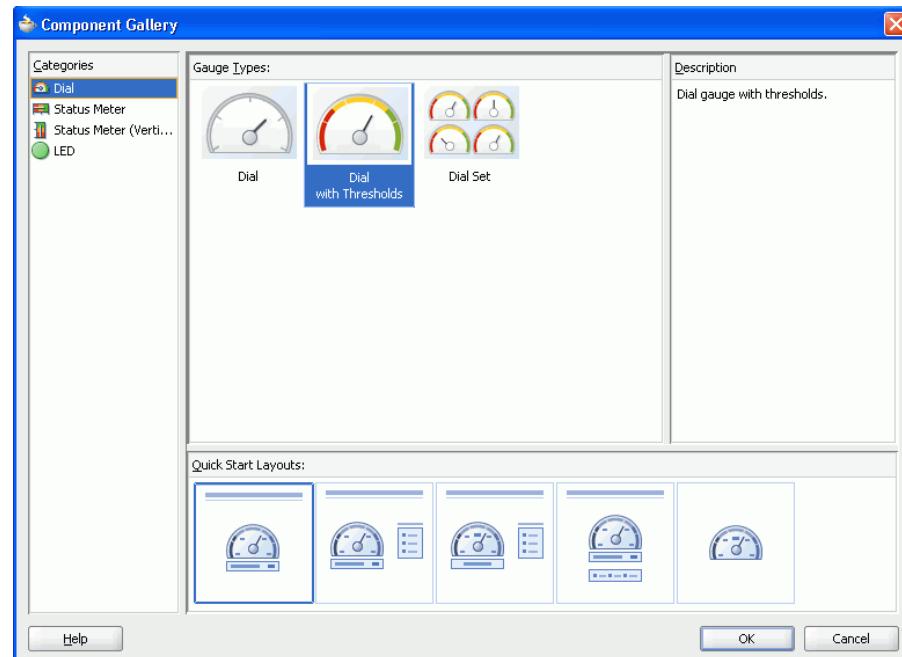
A gauge plots one data point with indication of whether the data point falls in an acceptable or unacceptable range. One databound gauge component can create a single gauge or an entire set of gauges, depending on the number of rows in the data collection used. In a collection, each row contains the values for a single gauge.

The Component Gallery for gauges allows you to choose from the following categories of gauges:

- **Dial:** Indicates the metric value of a task along a 180 degree arc.
- **Status Meter:** Indicates the progress of a task or the level of some measurement along a rectangular bar.
- **Status Meter (Vertical):** Indicates the progress of a task or the level of some measurement along a rectangular bar.
- **LED:** Depicts graphically a measurement such as a key performance indicator (KPI). Several styles of graphics are available for LED gauges such as arrows that indicate good (up arrow), fair (left- or right-pointing arrow), and poor (down arrow).

Each of these categories contains a number of different types of gauge. Explore the Component Gallery that appears when you create a gauge to view all available gauge and category types, and descriptions for each one. [Figure 24–6](#) shows the Component Gallery that appears for ADF gauges.

Figure 24–6 ADF Gauges Component Gallery



The data binding process is essentially the same regardless of which type of gauge you create. Only the metric value (that is, the measurement that the gauge is to indicate) is

required. However, if a row in a data collection contains range information such as maximum, minimum, and thresholds, then these values can be bound to the gauge to provide dynamic settings. If information that you want to use in a gauge's upper or lower labels is available in the data collection, then you can bind these values to the gauge also.

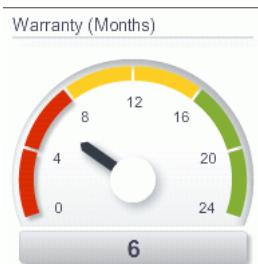
For information about customizing a gauge after the data binding is completed, see the "Using ADF Gauge Components" chapter in *Oracle Fusion Middleware Web User Interface Developer's Guide for Oracle Application Development Framework*.

24.3.1 How to Create a Databound Dial Gauge

You can use the ADF gauge component to create a dial gauge against a background that specifies ranges of values (also called *thresholds*) that vary from poor to excellent. The gauge indicator specifies the current value of the metric while the graphic allows you to evaluate the status of that value easily.

[Figure 24–7](#) shows a single dial gauge that appears if you create a gauge from the collection that stores `WarrantyPeriodMonths` data. Because only one gauge appears, this data collection must contain only a single row of data. The value of the metric (which is 6) appears in a label below the gauge. The range of values in the gauge is displayed as 0 to 24. Threshold ranges are identified at 8, 16, and 24 and are filled with the colors red for poor (below 8), yellow for adequate (from 8 to 16), and green for superior (above 16).

Figure 24–7 The Warranty in Months Dial Gauge



To create a dial gauge using a data control, you bind the gauge component to a collection. JDeveloper allows you to do this declaratively by dragging and dropping a collection from the Data Controls panel.

To create a databound dial gauge:

- From the Data Controls panel, select a collection.

For example, to create a dial gauge in the `StoreFrontModule` application to display the current warranty period in months for a product in a particular warehouse, you would select the `WarehouseStockLevels` collection.

[Figure 24–8](#) shows the `WarehouseStockLevels` collection in the Data Controls panel.

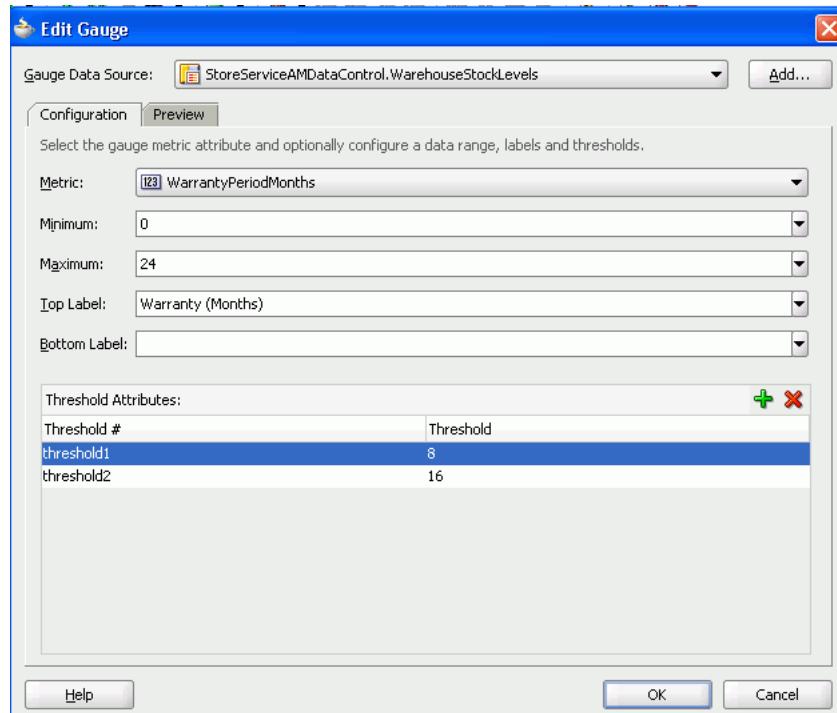
Figure 24–8 Data Collection with Warranty Period for a Product

2. Drag the collection onto a JSF page and, from the context menu, choose **Gauges**.
3. In the Component Gallery dialog, choose the category, type of gauge, and quick start layout, and then click **OK**.
4. In the Configuration page of the dialog, do the following:
 - In the **Metric** box, select the column in your data collection that contains the actual value that the gauge is to plot. This is the only required value in the dialog.
 - In the **Minimum** box, if your data collection stores a minimum value for the gauge range, select the column that contains this value.
 - In the **Maximum** box, if your data collection stores a maximum value for the gauge range, select the column that contains this value.
 - In the **Top Label** box, if your data collection stores a value that you want to display in the top label of the gauge, select the column that contains this value.
 - In the **Bottom Label** box, if your data collection stores a value that you want to display in the bottom label of the gauge, then select the column that contains this value.
 - In the **Threshold Attributes** list, if you want to specify threshold values, click the Add icon to insert a row for each threshold and specify the value in that row. Do not create a threshold equal to the maximum value for the gauge because the gauge automatically treats the maximum value as a threshold setting.
5. Optionally click the **Preview** tab to do the following:
 - Display a live preview of the gauge and its data
 - As needed, click the **Configuration** tab so that you can adjust the gauge specifications.
6. Click **OK**.

Figure 24–9 shows the dialog that appears if you examine the gauge binding for `WarehouseStockLevels` in the page definition file.

Note: The data source and metric data values are required. Other data values in the dialog are optional and can be specified in the gauge tag through the Property Inspector.

Figure 24–9 Edit Gauge Dialog



In the Property Inspector, after you complete the binding of the gauge, you can set values for additional attributes in the gauge tag and its child tags to customize the component.

24.3.2 What Happens When You Create a Dial Gauge from a Data Control

Dropping a gauge from the Data Controls panel has the following effect:

- Creates the bindings for the gauge and adds the bindings to the page definition file
- Adds the necessary code for the UI components to the JSF page

Example 24–3 shows the bindings that JDeveloper generated for the dial gauge that displays warranty in months for a product in a warehouse. This code example shows that the gauge metric receives its value dynamically from the `WarrantyPeriodMonths` column in the data collection and that the remaining data values have static settings.

Example 24–3 Bindings for a Dial Gauge

```
<gauge IterBinding="WarehouseStockLevelsIterator" id="WarehouseStockLevels"
       xmlns="http://xmlns.oracle.com/adfm/dvt">
  <gaugeDataMap>
    <item type="threshold" value="8" valueType="literal"/>
    <item type="threshold" value="16" valueType="literal"/>
    <item type="metric" value="WarrantyPeriodMonths"/>
```

```
<item type="minimum" value="0" valueType="literal"/>
<item type="maximum" value="24" valueType="literal"/>
<item type="topLabel" value="Warranty (Months)" valueType="literal"/>
</gaugeDataMap>
</gauge>
```

[Example 24–4](#) shows the code that JDeveloper generated in the JSF page for a dial gauge. The `<dvt:thresholdSet>` element specifies one `<dvt:threshold>` element for each threshold. Colors for the threshold ranges default to red, yellow, and green as specified by the values for the `fillColor` attributes. The `<dvt:indicator>` element specifies `IT_NEEDLE` as the indicator to use. This renders a needle at runtime. The default value for `<dvt:indicator>` renders a line (`IT_LINE`).

Example 24–4 Code on the JSF Page for an ADF Dial Gauge

```
<dvt:gauge id="gauge1"
    value="#{bindings.WarehouseStockLevels.gaugeModel}"
    gaugeType="DIAL" imageFormat="FLASH">
    <dvt:gaugeBackground>
        <dvt:specialEffects fillType="FT_GRADIENT">
            <dvt:gradientStopStyle/>
        </dvt:specialEffects>
    </dvt:gaugeBackground>
    <dvt:thresholdSet>
        <dvt:threshold text="Low" fillColor="#d62800"/>
        <dvt:threshold text="Medium" fillColor="#ffcf21"/>
        <dvt:threshold text="High" fillColor="#84ae31"/>
    </dvt:thresholdSet>
    <dvt:gaugeFrame/>
    <dvt:indicator type="IT_NEEDLE" />
    <dvt:indicatorBase/>
    <dvt:gaugePlotArea/>
    <dvt:tickLabel/>
    <dvt:tickMark/>
    <dvt:topLabel/>
    <dvt:bottomLabel/>
    <dvt:metricLabel position="LP_WITH_BOTTOM_LABEL" />
</dvt:gauge>
```

24.3.3 How to Create a Databound Status Meter Gauge Set

You can use the ADF gauge component to create a status meter gauge where the inner rectangle shows the current level of a measurement against the ranges marked in the outer rectangle. The graphic of the status meter gauge allows you to evaluate the condition or progress of a measurement easily.

[Figure 24–10](#) shows a set of status meter gauges that represent the inventory levels in a number of warehouses. This set of gauges results from binding one gauge component to a data collection (`WarehouseStockLevels`). This data collection contains a row of data for each warehouse. Each row produces one gauge in the set. Notice that all gauges in the set share the same range values of minimum (0) and maximum (1500) with thresholds set at 500 and 1000 and 1500. Each gauge in the set displays the name of the warehouse that it represents and the stock metric for that warehouse in its bottom label.

Figure 24–10 The Warehouse Inventory Status Meter Gauge Set

To create a status meter gauge set using a data control, you bind the gauge component to a data collection that contains multiple rows of data. JDeveloper allows you to do this declaratively by dragging and dropping a collection from the Data Controls panel.

To create a databound status meter gauge:

1. From the Data Controls panel, select a collection.

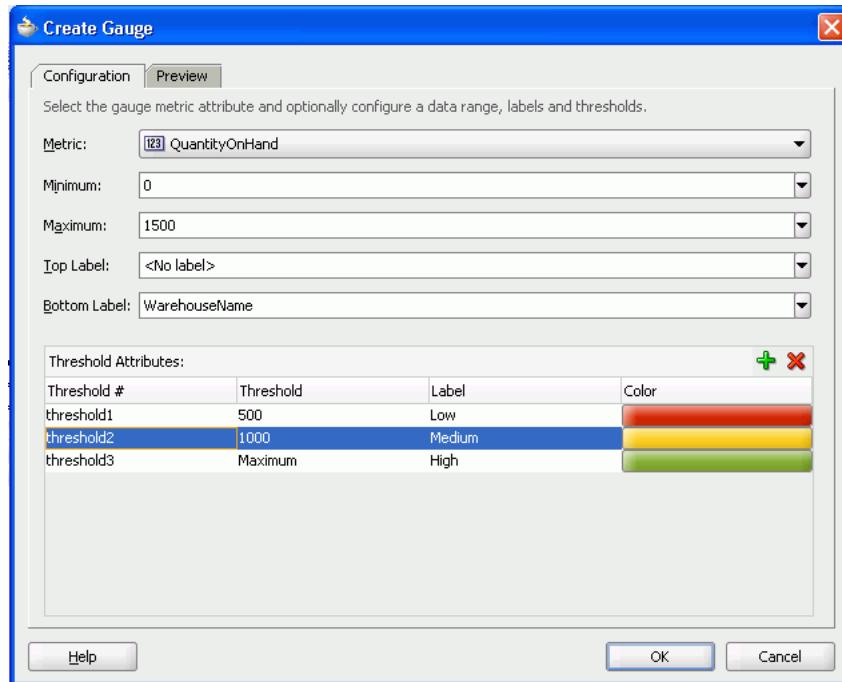
For example, to create a status meter gauge in the `StoreFrontModule` application that displays the quantity of stock on hand in a warehouse, you select the `WarehouseStockLevels` collection.

2. Drag the collection onto a JSF page and, from the context menu, choose **Gauges**.
3. In the Component Gallery, choose the following:
 - **Status Meter or Status Meter (Vertical)** in the **Categories** list
 - The type of gauge that you want to create
 - The quick start layout for the gauge at runtime
4. Click **OK**.
5. In the Create Gauge dialog that appears, select values as described in the following list:
 - Select an attribute binding from the **Metric** dropdown list. This attribute binding contains the actual value that the gauge is to plot.
 - Input a minimum value in the **Minimum** field if your data collection stores a minimum value for the gauge range.
 - Input a maximum value in the **Maximum** field if your data collection stores a maximum value for the gauge range.
 - Write or select a value in the **Top Label** field if you want to display a label on top of the gauge at runtime.
 - Write or select a value in the **Bottom Label** field if you want to display a label below the gauge at runtime.
 - Click the **Add** icon to insert a row for each threshold and specify the value for that threshold if you want to specify threshold values in the **Threshold Attributes** list. Do not create a threshold equal to the maximum value for the gauge because the gauge automatically treats the maximum value as a threshold setting.
6. Optionally, click the **Preview** tab to do the following:

- Display a live preview of the gauge and its data
- As needed, click the **Configuration** tab so that you can adjust the gauge specifications

[Figure 24–11](#) shows the settings for the set of status meter gauges that appears in [Figure 24–10](#). In addition to setting values for the required metric value, this dialog sets values for thresholds and for the name of the warehouse to appear in the gauge's bottom label.

Figure 24–11 Create Gauge Dialog for Warehouse Inventory Gauge Set



7. Click **OK**.

In the Property Inspector, after you complete the binding of the gauge, you can set values for additional attributes in the gauge tag and its child tags to customize the component as needed.

24.3.4 What Happens When You Create a Status Meter Gauge from a Data Control

Dropping a gauge from the Data Controls panel has the following effect:

- Creates the bindings for the gauge and adds the bindings to the page definition file
- Adds the necessary code for the UI components to the JSF page

[Example 24–5](#) shows the row set bindings that were generated for the status meter gauge set that displays inventory levels for each warehouse as illustrated in [Figure 24–11](#). This example shows the value binding created between the gauge metric attribute and the `QuantityOnHand` value in the data collection. It also shows the value binding between the `Bottom Label` attribute and the `WarehouseName` value in the data collection.

Example 24–5 Bindings Code for the ADF Status Meter Gauge Set

```
<gauge IterBinding="WarehouseStockLevelsIterator" id="WarehouseStockLevels"
       xmlns="http://xmlns.oracle.com/adfm/dvt">
  <gaugeDataMap>
    <item type="threshold" value="500" valueType="literal"/>
    <item type="threshold" value="1000" valueType="literal"/>
    <item type="metric" value="QuantityOnHand"/>
    <item type="minimum" value="0" valueType="literal"/>
    <item type="maximum" value="1500" valueType="literal"/>
    <item type="bottomLabel" value="WarehouseName"/>
  </gaugeDataMap>
</gauge>
```

Example 24–6 shows the code generated on the JSF page for the status meter gauge set that shows inventory levels for warehouses. The `gaugeSetColumnCount` attribute specifies that gauges should be displayed in two columns. The code also specifies three thresholds: Low, Medium, and High. For brevity, the value of the `inlineStyle` attribute has been omitted.

Example 24–6 Code on the JSF Page for the ADF Status Meter Gauge Set

```
<dvt:gauge id="gauge1"
            value="#{bindings.WarehouseStockLevels.gaugeModel}"
            gaugeType="STATUSMETER" imageFormat="FLASH">
  <dvt:gaugeBackground>
    <dvt:specialEffects fillType="FT_GRADIENT">
      <dvt:gradientStopStyle/>
    </dvt:specialEffects>
  </dvt:gaugeBackground>
  <dvt:thresholdSet>
    <dvt:threshold text="Low" fillColor="#d62800"/>
    <dvt:threshold text="Medium" fillColor="#ffcf21"/>
    <dvt:threshold text="High" fillColor="#84ae31"/>
  </dvt:thresholdSet>
  <dvt:indicatorBar/>
  <dvt:gaugePlotArea/>
  <dvt:tickLabel/>
  <dvt:tickMark/>
  <dvt:topLabel/>
  <dvt:bottomLabel/>
  <dvt:metricLabel position="LP_WITH_BOTTOM_LABEL" />
</dvt:gauge>
```

24.4 Creating Databound Pivot Tables

The ADF pivot table has the following structure:

- Column edge: You can specify one or more layers of information for the column edge of the pivot table. For example, a column edge might include the names of states or cities.
- Row edge: You can specify one or more layers of information for the row edge of the pivot table. For example, a row edge might include a product category and its products.
- Data body: Contains one or more measures (that is, data values) that you want to display in the cells of the pivot table. For example, you could include sales and costs in the data body of a pivot table.

[Figure 24–12](#) shows a pivot table that displays sales by state for all the products in each product category. This pivot table displays a row of totals at the end of each product category. [Figure 24–14](#) shows the data binding dialog that lays out the data for this pivot table and maps the items in the data control collection to the pivot table.

Figure 24–12 Sales Pivot Table by Product and State

		Sales									
		Colorado	Oregon	Wyoming	Idaho	California	New Mexico	Utah	Montana	Nevada	Washington
Audio and Video	Ipod Nano 1Gb	1499.5	1499.5	1499.5							
	Ipod Nano 2Gb				199.95						
	Ipod Nano 4Gb					499.9					
	Ipod Shuffle 1Gb						99.99				
	Ipod Speakers							89.99			
	Ipod Video 60Gb	399.99						399.99			
	LCD HD Television				899.99						
Cell Phones	Plasma HD Television	1999.99	1999.99	1999.99		1999.99			1999.99	1999.99	3999.9
	Tungsten E PDA		195.99								
	Zune 30Gb				225.99						
	Total	3899.48	3695.479!	3499.49	1099.94	2725.88	99.99	489.98	1999.99	1999.99	3999.9
	Bluetooth Adaptor						1999.99				1999.9
Games	Bluetooth Headset	49.99			49.99	149.97		49.99		49.99	49.9
	Treo 650 Phone/PDA	299.99	899.97	599.98		599.98	299.99	1199.96	299.99	599.98	299.9
	Total	349.98	899.97	649.97	149.97	599.98	2349.97000	1199.96	349.98	599.98	2349.97000
Games					179.99						

For information about customizing a pivot table after data-binding is completed, see the "Using ADF Pivot Table Components" chapter in *Oracle Fusion Middleware Web User Interface Developer's Guide for Oracle Application Development Framework*.

24.4.1 How to Create a Pivot Table

To create a pivot table using a data control, you bind the pivot table component to a collection. JDeveloper allows you to do this declaratively by dragging and dropping a collection from the Data Controls Panel.

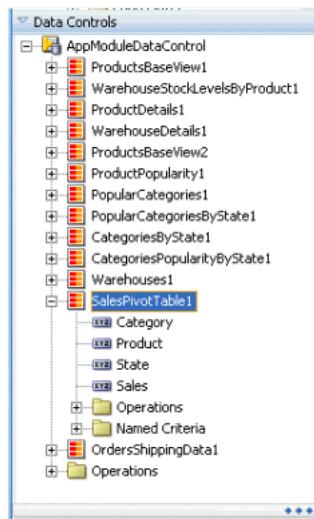
Tip: You can also create a pivot table by dragging a pivot table from the Component Palette and completing the Create Pivot Table dialog. This approach allows you the option of designing the pivot table user interface before binding the component to data.

To create a databound pivot table:

- From the Data Controls panel, select a collection.

[Figure 24–13](#) shows an example where you could select the SalesPivotTable1 collection in the Data Controls panel to create a pivot table that displays sales by state for each product.

Figure 24–13 Data Collection for Product Sales by State



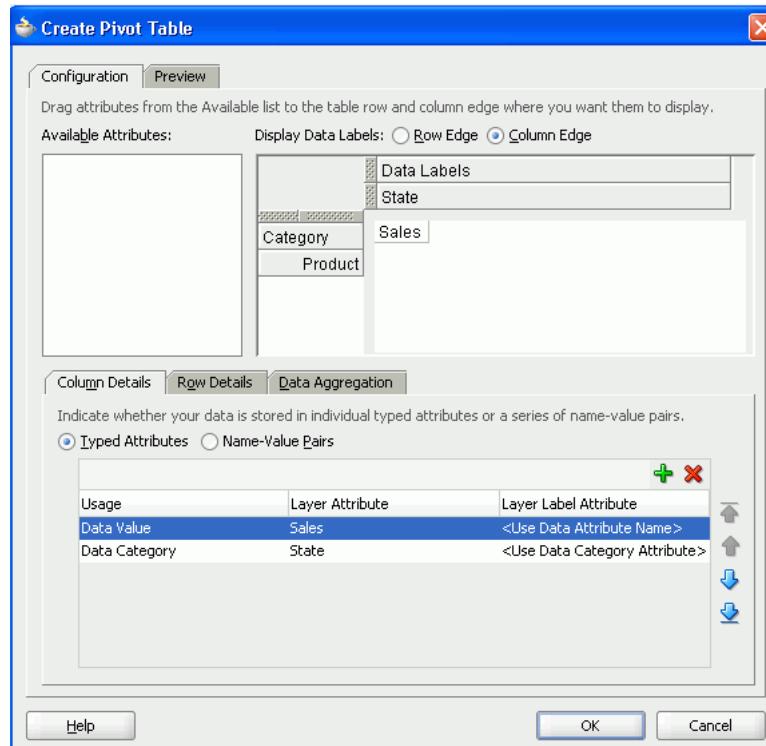
- Drag the collection onto a JSF page and, from the context menu, choose **Tables**, then **ADF Pivot Table**.
- In the ensuing Create Pivot Table dialog, do the following:
 - In the top of the Configuration page, select the attributes for the pivot table's columns, rows, and data body by dragging the attributes from the **Available Attributes** list to the pivot table layout beside the list.

For example, to lay out the sales pivot table shown in [Figure 24–12](#), do the following: drag the State attribute to the column edge, drag the Product attribute to the row edge, and drag the Sales attribute to the data body in the pivot table so that Sales appears in the Data Labels layer. [Figure 24–14](#) shows the resulting Create Pivot Table dialog.

Note: Data Labels refers to a layer of the pivot table that identifies the data in the cells. Items that you drag to the data body of the pivot table appear in the Data Labels layer.

You can drag Data Labels to any location on the column edge or the row edge. You can also drag attributes to different locations on the same edge or on another edge.

As you lay out the pivot table in the upper portion of the Configuration page, corresponding entries are created in the Column Details page and the Row Details page of the data binding dialog. Similarly, changes that you make to Column Details or Row Details pages affect the layout of the pivot table shown at the top of the Configuration page.

Figure 24–14 Create Pivot Table Dialog with Layout and Column Details

- b. To examine and possibly edit information about columns, use the Column Details page at the bottom of the Configuration page. In this page, you can specify additional information regarding the attributes that appear in the column edge of the pivot table.

To specify alternative labels for items in a pivot table column layer, select the attributes that contain the alternative label in the **Layer Label Attribute** column. Optionally, you can use the arrow icons on the side of the Column Details page to rearrange the sequence of the items in the column layers to affect the layout of the pivot table column edge.

- c. To examine and possibly edit information about rows, click **Row Details**. In the Row Details page in the lower portion of the Configuration page, you can specify additional information regarding the attributes that appear in the row edge of the pivot table.

To specify alternative labels for items in a pivot table row layer, select the attributes that contain the alternative label in the column entitled **Layer Label Attribute**. Optionally, you can also use the arrow icons on the side of the Row Details page to rearrange the sequence of the items in the row layers to affect the layout of the pivot table row edge.

4. If you want to define totals for the pivot table, click **Data Aggregation**. To specify totals do the following:

- a. Do *not* clear the **Enable duplicate row aggregation** checkbox if there is any possibility that data values from duplicate rows in the data collection will apply to a single cell in the pivot table.

For additional information about aggregating duplicate rows of data, see [Section 24.4.3, "What You May Need to Know About Aggregating Attributes in the Pivot Table"](#).

- b. For each attribute that you want to total, click the **Add** icon to create a row to define the desired total.
- c. In the **Attribute** column, select the attribute that you want to total.

For example, to create a pivot table that totals product sales, like the pivot table shown in [Figure 24–12](#), select Product as the attribute to total.

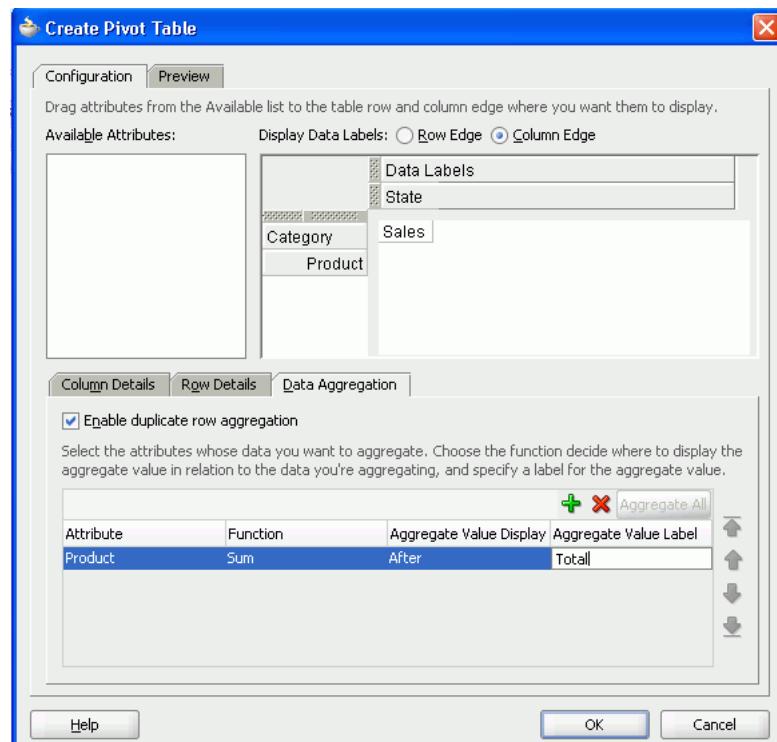
As an additional example, consider what happens if state were swapped with category in the layout of the pivot table shown in [Figure 24–12](#). In that case, a total of product sales would appear after each set of products for a state.

- d. In the **Function** column, select the mathematical operation that you want to use for the aggregation. Available options are Sum, Average, Count, Maximum, Minimum, Standard Deviation, and Variance.
- e. In the **Aggregate Value Display** column, select the value that indicates where you want the aggregate display to appear relative to the item mentioned in the Attribute column. Valid values are: Before, After, or Replace.
- f. In the **Aggregate Value Label** column, enter the text that you want to use as a label for the aggregation.

For example, you might want to use the text "Total" as the aggregation label.

[Figure 24–15](#) shows the Data Aggregation page for a pivot table that totals product sales and displays the total after each set of products.

Figure 24–15 Data Aggregation Page of Pivot Table Data Binding Dialog



- g. Click **Preview** to see the data that will be displayed in the pivot table. The preview does not require that you compile and run code. If you are not satisfied with the preview, alter the settings in the binding dialog and return again to the Preview page to verify that your data looks as desired.

Figure 24–16 shows the Preview page for the pivot table that displays product sales for each state with totals after each set of products.

Figure 24–16 Live Data Preview of Pivot Table

The screenshot shows the 'Create Pivot Table' dialog box with the 'Preview' tab selected. The dialog has a title bar with a close button and two tabs: 'Configuration' and 'Preview'. Below the tabs is a table with the following data:

		Sales			
		Colorado	Oregon	Wyoming	Id
Audio and Video	Ipod Nano 1Gb	1,500	1,500	1,500	
	Ipod Nano 2Gb				
	Ipod Nano 4Gb				
	Ipod Shuffle 1Gb				
	Ipod Speakers				
	Ipod Video 60Gb	400			
	LCD HD Television				
	Plasma HD Television	2,000	2,000	2,000	
	Tungsten E PDA		196		
	Zune 30Gb				
Total	3,899	3,695	3,499		
Cell Phones	Bluetooth Adaptor				
	Bluetooth Phone Headset	50		50	
	Treo 650 Phone/PDA	300	900	600	
	Total	350	900	650	
Games	Nintendo DS				

At the bottom of the dialog are 'Help', 'OK', and 'Cancel' buttons.

24.4.2 What Happens When You Use the Data Controls Panel to Create a Pivot Table

Dropping a pivot table from the Data Controls panel has the following effect:

- Creates the bindings for the pivot table and adds the bindings to the page definition file
- Adds the necessary code for the UI components to the JSF page

24.4.2.1 Bindings for Pivot Tables

Example 24–7 shows the row set bindings that were generated for the pivot table that displays product sales within category by state. Notice that the pivot table data map contains a columns element that shows each column item, a rows element that includes each row item in the appropriate sequence, and an aggregated items element that defines the aggregation.

In the data binding dialog for this pivot table, the **Enable duplicate row aggregation** control was selected. For this reason, the <data> element shows that aggregateDuplicates is set to True. For more information about aggregating duplicates, see Section 24.4.3, "What You May Need to Know About Aggregating Attributes in the Pivot Table".

Example 24–7 Binding XML for the ADF Pivot Table

```
<pivotTable IterBinding="SalesPivotTable1Iterator" id="SalesPivotTable1"
            xmlns="http://xmlns.oracle.com/adfm/dvt">
    <pivotTableDataMap>
```

```

<columns>
    <data aggregateDuplicates="true">
        <item value="Sales"/>
    </data>
    <item value="State"/>
</columns>
<rows>
    <item value="Category"/>
    <item value="Product"/>
</rows>
<aggregatedItems>
    <item aggregateLocation="After" aggregateType="Sum"
          value="Product" aggregateLabel="Total"/>
</aggregatedItems>
</pivotTableDataMap>
</pivotTable>

```

24.4.2.2 Code on the JSF Page for an ADF Pivot Table

[Example 24–8](#) shows the code generated on the JSF page for the sales pivot table. The only customization used in this code is the setting of the `inlineStyle` attribute in the pivot table tag. Instead of the default setting of `xxx` pixels, the width for the sales table is set to 800 pixels.

Example 24–8 XML Code on a JSF Page for the ADF Pivot Table

```

<dvt:pivotTable id="pivotTable1"
    inlineStyle="width:800px"
    value="#{bindings.SalesPivotTable1.pivotTableModel}"/>

```

24.4.3 What You May Need to Know About Aggregating Attributes in the Pivot Table

If the attributes that you choose to display in your pivot table do not uniquely identify each row in your data collection, then you can aggregate the data from duplicate rows to collapse that data into a single pivot table cell.

For example, if the rows in the data collection shown in [Figure 24–12](#) also contained a store identification, then the data rows from all stores in a given combination of Product, Category, and State would have to be collapsed into a single cell in the pivot table. For accurate calculation of such duplicate rows, you must ensure that the **Enable duplicate row aggregation** control is selected in the Data Aggregation page of the Create Pivot Table dialog.

The pivot table has the following optional data binding attributes available for controlling the calculation of duplicate data rows:

- `aggregateDuplicates`: Boolean property of the `<data>` element that determines whether special processing is enabled at binding runtime to aggregate data values in duplicate rows. If this attribute is not specified, then `false` is assumed.
- `defaultAggregateType`: String property of the `<data>` element that specifies a default aggregation method for handling duplicates. Valid values are `SUM`, `AVERAGE`, `COUNT`, `MIN`, `MAX`, `STDDEV`, `VARIANCE`. If `aggregateDuplicates` is `true` and `defaultAggregateType` is unspecified, then `SUM` is assumed.
- `aggregateType`: String property of an `<item>` element that enables you to override the default aggregate type for a particular data item. This attribute is useful only when you have multiple data values (such as Sales and Expenses) bound to your pivot table.

24.4.3.1 Default Aggregation of Duplicate Data Rows

By default, the pivot table uses the SUM operation to aggregate the data values of duplicate data rows in a data collection to produce a single cell value in the pivot table. This means that the aggregateDuplicates attribute is set to true and the defaultAggregateType is assumed to be SUM.

The <data> element shown in [Example 24–7](#) is an example of such default aggregation.

24.4.3.2 Custom Aggregation of Duplicate Rows

If you want the pivot table to use a different mathematical operation to aggregate the data values of duplicate rows, then you set the defaultAggregateType to the desired operation.

[Example 24–9](#) shows a data element with the defaultAggregateType set to SUM. This operation would be appropriate if you want to see the total of sales from all stores for each unique combination of Product, Category, and State.

Example 24–9 Binding XML for Custom Aggregation of Duplicate Rows

```
<pivotTable IterBinding="SalesPivotTable1Iterator" id="SalesPivotTable11"
    xmlns="http://xmlns.oracle.com/adfm/dvt">
    <pivotTableDataMap>
        <columns>
            <data aggregateDuplicates="true" defaultAggregateType="SUM">
                <item value="Sales"/>
            </data>
            <item value="State"/>
        </columns>
        <rows>
            <item value="Category"/>
            <item value="Product"/>
        </rows>
        <aggregatedItems>
            <item aggregateLocation="After" aggregateType="AVERAGE"
                value="Product" aggregateLabel="Average"/>
        </aggregatedItems>
    </pivotTableDataMap>
</pivotTable>
```

If you have a pivot table with multiple data values (such as sales and the average size of a store in square feet) and you want to sum the sales data values in duplicate rows, but you want to average the square feet data values, then do the following:

- On the <data> element, set the defaultAggregateType to SUM.
- On the <item> element for the square feet attribute, set the aggregateType to AVERAGE.

[Example 24–10](#) shows the <columns> elements wrapped by a PivotTableDataMap element. The <data> element contains the default attributes for aggregation. These apply to all data items that do not have a specific custom aggregateType attribute specified.

Example 24–10 Data and Item Elements for Multiple Custom Aggregations

```
<columns>
    <data aggregateDuplicates="true" defaultAggregateType="SUM">
        <item value="Sales" label="Total Sales"/>
        <item value="StoreSqFeet" label="Avg Sq Feet" aggregateType="AVERAGE"/>
    </data>
</columns>
```

```

</data>
<item value="State"/>
</columns>

```

24.4.4 What You May Need to Know About Specifying an Initial Sort for a Pivot Table

By default, a pivot table initially sorts data based on values in the outer row labels. You can change the default behavior by inserting a `sorts` tag inside the `pivotTableDataMap` tag of a pivot table binding in the page definition file. Insert a `qdrSliceSort` tag inside the `sorts` tag and set values for the attributes as described in [Table 24–2](#).

Table 24–2 Attribute Values for `qdrSliceSort` Tag

Attribute	Description
<code>direction</code>	Specify the initial direction of the sort. Valid values are <code>ASCENDING</code> and <code>DESCENDING</code> . A value for this attribute is required.
<code>edge</code>	Specify <code>columns</code> or <code>rows</code> to determine which edge sorts data. A value for this attribute is required.
<code>grouped</code>	Specify <code>true</code> if you want to sort slices within their parent or <code>false</code> if you want to sort across the entire edge. A value for this attribute is optional. The default value is <code>false</code> .
<code>nullsFirst</code>	Specify <code>true</code> if you want null values to appear at the beginning of a sort and <code>false</code> if you want null values to appear at the end of a sort. A value for this attribute is optional. The default value is <code>false</code> .

Insert one or more `item` tags inside the `qdrSliceSort` tag. An item tag specifies the slice on the opposite edge from which the values to be sorted should be obtained. Set values for the attributes as described in [Table 24–3](#).

Table 24–3 Attribute Values for `item` Tag

Attribute	Description
<code>name</code>	Specify the name of the layer to sort on. Typically, this is the column name in the row set. Specify <code>DataLayer</code> if you want to specify the layer that contains the data columns in a row set (for example, Sales, Costs, and so on).
<code>value</code>	Specify the value of the specified layer on the desired slice.

24.5 Creating Databound Geographic Maps

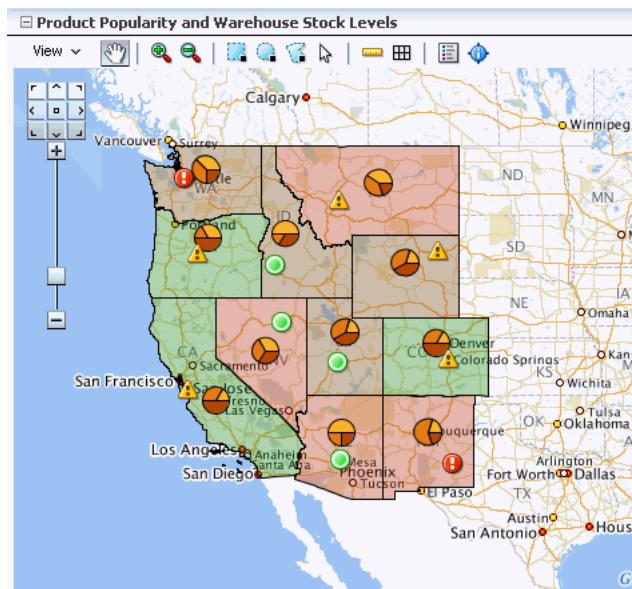
An ADF geographic map is an ADF Data Visualization component that provides the functionality of Oracle Spatial within Oracle ADF. This component allows users to represent business data on a geographic map and to superimpose multiple layers of information (known as *themes*) on a single map. These layers can be represented as any of the following themes: bar graph, pie graph, color, point, and predefined theme.

[Figure 24–17](#) shows a geographic map component that uses a base map for a region in the United States with the following themes:

- Color theme: For the selected product, this theme colors states based on product popularity. The colors range from green (which represents the highest popularity for that product) to red (which represents the lowest popularity for that product).
- Pie graph theme: This theme displays a pie graph in each state to indicate the popular product categories in that state. In this example, the pie graph shows the following product categories as pie slices: Media, Office, and Electronics.

- Point theme: This theme identifies warehouses as points. For each point, it displays an icon to indicate the inventory level at that warehouse for the selected product. A separate icon is displayed for each of the following ranges of inventory: low inventory, medium inventory, and high inventory.

Figure 24–17 Geographic Map with Color Theme, Pie Graph Theme, and Point Theme for a Product



A geographic map component differs from other ADF Data Visualization components as you cannot put multiple maps on a page. This contrasts to components such as graphs where you can put multiple graphs on a page. Instead, you show how multiple sets of data relate to each other spatially or, for a specific point, you display different attributes layered in separate themes.

The geographic map component itself is not bound to data. However, each map *theme* has its own data bindings.

A base map forms the background on which the ADF geographic map component layers the themes that developers create.

In Oracle Spatial, administrators create base maps that consist of one or more themes. The administrator controls the visibility of the base map themes. When you zoom in and out on a base map, various base map themes are hidden or displayed. At the ADF geographic map component level, you cannot use zoom factor to control the display of the themes created by the administrator on the base map.

When you overlay themes on the ADF geographic map, you can control the visibility of your themes by setting the maxZoom and minZoom attributes of the tags related to these themes. At runtime, you can also hide or display your custom themes by using the View menu of the Map toolbar or by using other ADF components that you create on the page.

For information about customizing a geographic map after data-binding is completed, see the "Using ADF Geographic Map Components" chapter in *Oracle Fusion Middleware Web User Interface Developer's Guide for Oracle Application Development Framework*.

24.5.1 How to Create a Geographic Map with a Point Theme

To create a geographic map, you first configure the map (that is, select a base map and provide URLs for processing) and then bind a theme of the map to a data collection. JDeveloper allows you to do this declaratively by dragging and dropping a collection from the Data Controls panel for the theme that you want to create.

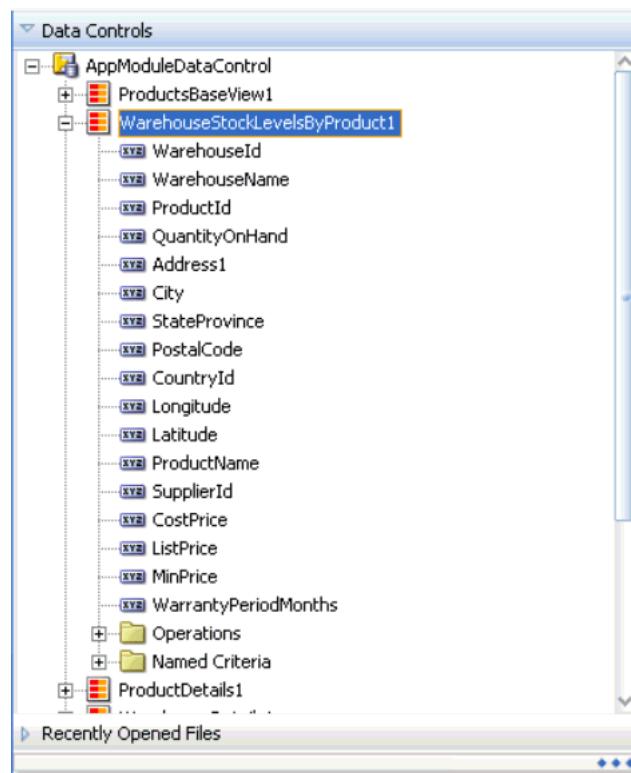
When you create a map point theme, you have the option of customizing the style of the points that appear in the map. For each different point style, you can use a `mapPointStyleItem` tag.

To create a geographic map with a databound point theme:

- From the Data Controls panel, select a collection.

Figure 24–18 shows an example where you could select the `WarehouseStockLevelsByProduct1` collection in the Data Controls panel to create a geographic map with a point theme that displays an image to represent the quantity on hand for each warehouse point.

Figure 24–18 Data Collection for Warehouse Stock Levels



- Drag the collection onto a JSF page and, from the context menu, choose **Geographic Map**, then **Map and Point Theme**.
- If you have not yet configured a map on the page, then in the ensuing Create Geographic Map dialog, click the **New** icon to display the Create Geographic Map Configuration dialog and do the following:
 - In the **Id** field enter the unique identifier for the map configuration.
 - In the **MapView URL** field enter the URL for the Oracle Application Server MapViewer Service.

- c. In the **Geocoder URL** field select the URL for the Geocoder Web service that converts street addresses into latitude and longitude coordinates for mapping.

Note: The Geocoder URL is needed only if you do *not* already have longitude and latitude information for addresses.

- d. Click **OK** to dismiss the dialog and return to the Create Geographic Map dialog.
4. In the Maps page, you select the base map for the geographic map component and provide other settings to use with the map by doing the following:
 - a. From the **Data Source** list select the collection of maps from which you will choose a base map.
 - b. From the **Base Map** list select the map that will serve as the background for the geographic map component.
 - c. To specify values for the **StartingX** field and the **StartingY** field click on the image of the map to center it within the Preview window.
You can use the arrows in the map navigator in the upper left-hand corner to move the map in the appropriate direction.
 - d. Optionally use the sliding arrow in the Preview window to adjust the zoom factor of the map.
 - e. Click **OK** to dismiss the dialog and to display the Create Point Map Theme dialog.
5. In the **Theme Id** field enter the unique identifier for the point map theme.
6. In the **Location** section, specify whether the point location is to be specified by a pair of *x* and *y* coordinates (longitude and latitude) or as an address.
The choice you select for location will determine which controls appear in the Location section.

Tip: Using *x* and *y* coordinates is a more efficient way to present data on the map rather than using the Address controls, which must be converted by a Geocoder to *x* and *y* coordinates. If the data collection has more than 100 rows, then to ensure adequate performance, use *x* and *y* coordinates.

7. For the *x* and *y* point location, you select the data that corresponds to the following items:
 - **X (Longitude):** The horizontal location of the point on the map.
 - **Y (Latitude):** The vertical location of the point on the map.
 - **Label:** The labels for the points in the top section of the information window, which is displayed when you click a point.
8. In the Point Data section, provide the following information that identifies the data associated with the point, its label, and optionally the style for the point:
 - In the **Data** field, select the data column that is associated with the point, such as *QuantityOnHand*.
 - In the **Label** field, enter the text that will appear in the information window before the data value when you click a point.

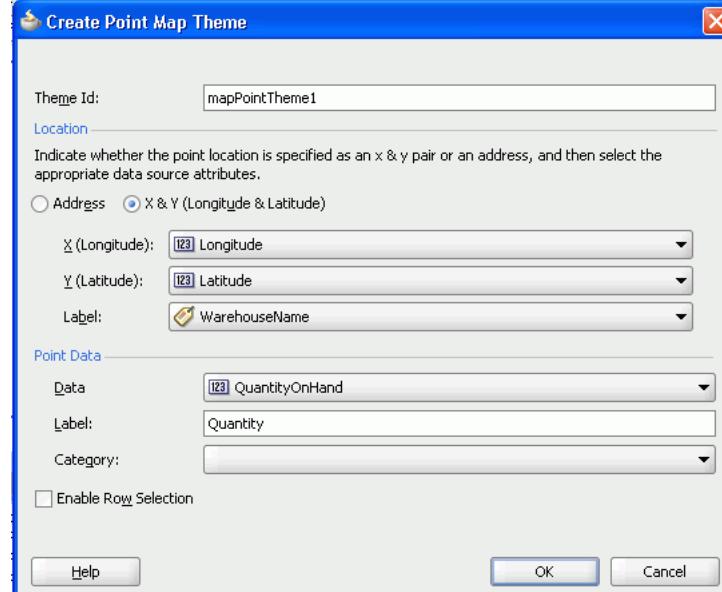
- Optionally, in the **Category** field, select a data column to use for finding the appropriate style for a point. If you select a value for Category, that value is stored in the binding for this point theme and then matched against the `itemValue` attribute of the `mapPointStyleItem` tags that you create for this point theme.

Note: If your data does not have a column that you want to use as a category for finding the style of a point, you can also use `mapPointStyleItem` tags to define styles related to data ranges (such as high, medium, and low) that are matched to the values in the column that you select in the Data field. For more information, see [Section 24.5.2, "How to Create Point Style Items for a Point Theme"](#).

9. Select the **Enable Row Selection** field only if you want to enable the selection of rows in a related component. You select this option when the page contains a component that is linked to a data collection which is related to the geographic map that you are creating.
10. Click **OK**.

[Figure 24–19](#) shows the Create Point Map Theme dialog for a geographic map with a point theme that displays an image representing quantity on hand for each warehouse point.

Figure 24–19 Create Point Map Theme Dialog for Warehouse Inventory Levels



24.5.2 How to Create Point Style Items for a Point Theme

There are a variety of options available for creating point style items for use in a given map point theme. These are:

- A single image for all data points
- Separate images for each data point category
- Images that represent low, medium, and high data value ranges

After you create the data binding for a map point theme, you have the option of selecting a single built-in image that should be used for all points in that map theme. In the Property Inspector, you can make this selection in the `builtInImage` attribute of the `mapPointTheme` tag. The default value for this attribute is `OrangeBall`.

Alternatively, if you specify a value for `Category` in the Create Point Map Theme dialog, then you should also create a set of point style items to determine a separate image that represents data points in each category. In this case, you do not use the minimum and maximum values in the point style item tags. Instead, you set the `itemValue` attribute of point style item tags to a value that matches entries in the data column that you specified for `Category`.

In a point theme for a geographic map, if you do not specify a value for `Category`, you can still use the `mapPointStyleItem` child tags of the `mapPointTheme` tag to specify ranges of values (such as low, medium, and high) and the images that are to represent these ranges. If you do this, then each point will be represented by an image that identifies the range in which the data value for that point falls.

The following procedure assumes that you have already created a geographic map with a point theme.

To add point style items to a map point theme to represent low, medium, and high data value ranges:

1. In the Structure window, right-click the `dvt:mapPointTheme` tag and choose **Insert inside the dvt:mapPointTheme, then Point Style Item**.
2. In the Point Style Item Property Inspector, set values as described [Table 24–4, "Properties for Point Style Item"](#).

Table 24–4 Properties for Point Style Item

For this property...	Set this value...
Id	Specify a unique ID for the point style item.
MinValue	Specify the minimum value in a data range that you define.
MaxValue	Specify the maximum value in a data range that you define.
ShortLabel	Specify text to appear when a user hovers over the point item. For example, if you define a point item for low inventory, then enter <code>Low Inventory</code> as the value for this property.
ImageURL	Specify the URL to the image file or select it from the dropdown list. At runtime, the image you specify appears on the map to represent the data range identified by the MinValue and MaxValue properties. Alternatively, you can select one of a number of predefined images referenced by the <code>BuiltInImage</code> dropdown list that appears in the Other section.
Hover imageURL	Specify the URL to the image file or select it from the dropdown list. At runtime, the image you specify appears when a user hovers over the point item.
Selected imageURL	Specify the URL to the image file or select it from the dropdown list. At runtime, the image you specify appears when a user selects the point item.

3. If you defined a data value range for a low data value range in Steps 1 and 2, then repeat Steps 1 and 2 to define medium and high data value ranges with appropriate values.

Note: The use of `mapPointStyleItem` child tags to customize the style of points is a declarative approach that lets you provide custom point images. For information about using a callback to provide not only custom images but also custom HTML, see [Section 24.5.4, "What You May Need to Know About Adding Custom Point Style Items to a Map Point Theme"](#).

24.5.3 What Happens When You Create a Geographic Map with a Point Theme

Dropping a geographic map and a point theme (which in this case would be the initial theme added to the map) from the Data Controls panel has the following effect:

- Creates the bindings for the point theme and adds the bindings to the page definition file
- Adds the necessary tags to the JSF page for the geographic map component
- Adds the necessary point theme tags to the JSF page within the map XML

24.5.3.1 Binding XML for a Point Theme

[Example 24–11](#) shows the row set bindings that were generated for the point theme of the geographic map.

Example 24–11 Point Theme Binding XML

```
<mapTheme IterBinding="WarehouseStockLevelsByProduct1Iterator"
    id="WarehouseStockLevelsByProduct1"
    xmlns="http://xmlns.oracle.com/adfm/dvt">
    <mapThemeDataMap mapThemeType="point">
        <item type="data" value="QuantityOnHand" label="Product Quantity"/>
        <item type="lat_long" latitude="Latitude"
            longitude="Longitude" label="WarehouseName" />
    </mapthemeDataMap>
</mapTheme>
```

24.5.3.2 XML Code on the JSF Page for a Geographic Map and Point Theme

[Example 24–12](#) shows the XML code generated on the JSF page for the geographic map and its point theme. Notice the code for the three kinds of point style settings based on data value.

The initial point style setting (`ps0`) applies to values that do not exceed 500. This point style displays an image for very low inventory and provides corresponding tooltip information.

The second point style setting (`ps1`) applies to values between 500 and 1000. This point style displays an image for low inventory and provides corresponding tooltip information.

The final point style setting (`ps2`) applies to values between 1000 and 1600. This point style displays an image for high inventory and provides corresponding tooltip information.

Example 24–12 Geographic Map and Point Theme XML Code on the JSF Page

```
<dvt:map id="map1"
    mapServerConfigId="mapConfig1"
    inlineStyle="width:850px;height:490px"
    startingX="-96.0"
```

```

baseMapName="ELOCATION_MERCATOR.WORLD_MAP"
startingY="37.0" zoomBarPosition="WEST"
showScaleBar="false"
partialTriggers="go pointTheme pieTheme colorTheme" mapZoom="3">
    <dvt:mapPointTheme id="mapPointTheme1"
        shortLabel="Warehouse Stock Levels"
        selectionListener="#{MapBean.processSelection}"
        value="#{bindings.WarehouseStockLevelsByProduct1.geoMapModel}"
        rendered="#{AppState.showPointTheme}">
        <dvt:mapPointStyleItem id="ps0"
            minValue="0"
            maxValue="500"
            imageURL="/images/low.png"
            selectedImageURL="/images/lowSelected.png"
            shortLabel="Very Low Inventory"/>
        <dvt:mapPointStyleItem id="ps1"
            minValue="500"
            maxValue="1000"
            imageURL="/images/medium.png"
            selectedImageURL="/images/mediumSelected.png"
            shortLabel="Low Inventory"/>
        <dvt:mapPointStyleItem id="ps2"
            minValue="1000"
            maxValue="1600"
            imageURL="/images/regularGreen.png"
            selectedImageURL="/images/regularGreenSelected.png"
            shortLabel="High Inventory"/>
    </dvt:mapPointTheme>
</dvt:map>

```

24.5.4 What You May Need to Know About Adding Custom Point Style Items to a Map Point Theme

If you want to provide custom HTML as well as custom images for map points, then you can use the `customPointCallback` attribute of the `dvt:mapPointTheme` tag to accomplish this customization.

Important: If you set the `customPointCallback` attribute for a map point theme, the map ignores any `dvt:mapPointStyleItem` child tags because the callback overrides these tags.

To use a callback to customize the style of map points:

1. Write a method in Java to perform the desired point customization.
2. Store this method in a managed bean for the map.

For more information about managed beans, see the "Creating and Using Managed Beans" section in the *Oracle Fusion Middleware Web User Interface Developer's Guide for Oracle Application Development Framework*.

3. After you finish data-binding the map point theme, use the Property Inspector to specify a reference to the managed bean method in the `customPointCallback` attribute of the `dvt:mapPointTheme` tag.

For example, if the managed bean is named `MapSampleBean` and the method is named `setCustomPointStyle`, then the reference becomes
`# {mapSampleBean.CustomPointStyle}`.

24.5.5 How to Add a Databound Color Theme to a Geographic Map

When you create a geographic map, you can choose to create themes (point, color, and graph) in any sequence that you wish.

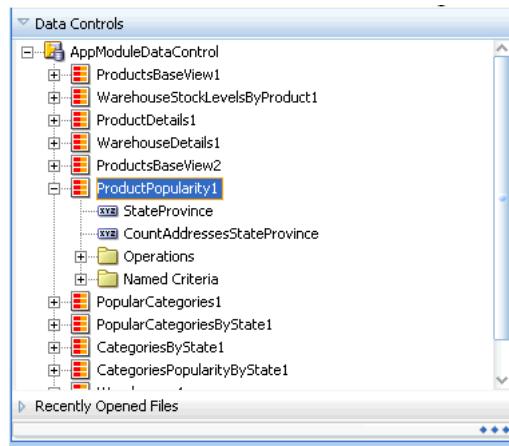
The following procedure assumes that a geographic map has already been configured and, therefore, the map component does not display the dialog for configuring the map. Instead, only the dialog for creating the color theme appears.

To add a databound color theme to a geographic map:

- From the Data Controls panel, select a collection.

[Figure 24–20](#) shows an example where you could select the ProductPopularity1 collection in the Data Controls panel to create a color map theme that shows product popularity by the color of regions (for example, states).

Figure 24–20 Data Collection for Product Popularity by State

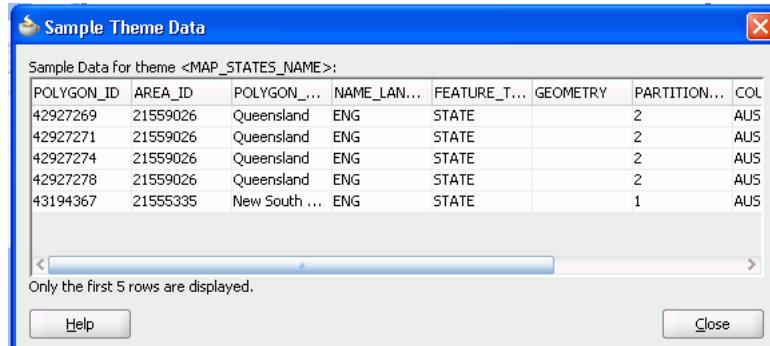


- Drag the collection onto a JSF page which already contains a geographic map component and, from the context menu, choose **Geographic Map**, then **Color Theme**.
- In the ensuing Create Color Map Theme dialog, enter a unique identifier for the map theme in the **Id** field.
- In the **Base Map Theme** section, identify the base map color theme to use for the geographic map by doing the following:
 - In the **Name** field, select the name of the base map theme.
 - For **Location**, select the location column in the data collection that should be matched to the location column in the base map theme that you selected.
 - Optionally, click **View Sample Theme Data** to display the Sample Theme Data dialog, in which you can examine the first several rows of the actual data so that you can identify the appropriate location column.

For example, if you want to view the data for a region that consists of states in the United States map, you might select MAP_STATES_NAME as shown in [Figure 24–21](#).

Note: It is possible for an administrator of Oracle Spatial to disable the display of sample data. If this button is not available, then consult the administrator for guidance.

Figure 24–21 Sample Theme Data for Regions or States

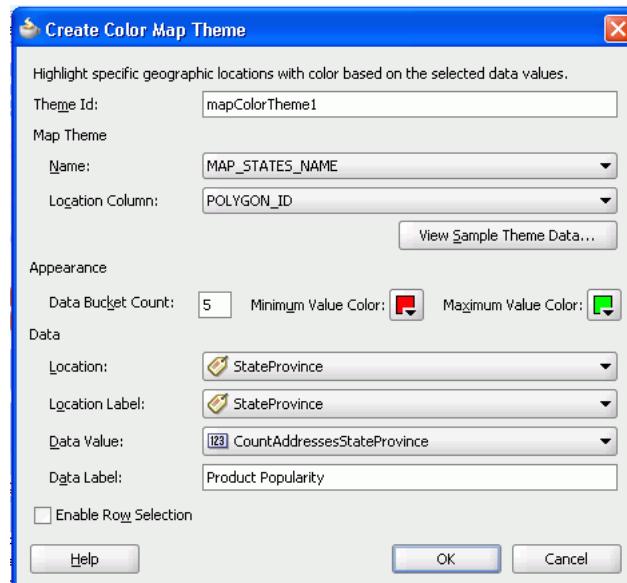


5. In the **Appearance** section, specify the look of the color theme as follows:
 - a. In **Data Bucket Count**, enter the number of groups for the data in this geographic map. Each group is coded with a color. After specifying this number, you can provide colors for the minimum value and the maximum value. The colors for the other values are chosen automatically using an RGB algorithm.
 - b. In **Minimum Value Color**, select the color for the minimum value.
 - c. In **Maximum Value Color**, select the color for the maximum value.

Note: If you want to specify an exact color for each data bucket, see [Section 24.5.7, "What You May Need to Know About Customizing Colors in a Map Color Theme"](#).

6. In the **Data** section, provide the following information about the data in the collection:
 - a. For **Location**, select the column in the data collection that should match the values in the location column that you selected from the base map theme.
 - b. For **Location Label**, select the column in the data collection that contains the labels associated with the values in the location column. These labels are shown in the information window that is displayed when you click or hover over a color.
 - c. For **Data Label**, enter the label to use for describing the data in the information window and the tooltip that is displayed when you click or hover over a color. For example, the information window might include a label before the data value, such as **Product Popularity**.
7. Use **Enable Row Selection** only if you want to enable master-detail relationships. This is useful when the data collection for the map theme is a master in a master-detail relationship with a detail view that is displayed in another UI component on the page.

Figure 24–22 shows the Create Color Map Theme dialog for the product popularity by state color theme.

Figure 24–22 Create Color Map Theme for Product Popularity By State

24.5.6 What Happens When You Add a Color Theme to a Geographic Map

Dropping a color theme from the Data Controls panel to an existing geographic map has the following effect:

- Creates the bindings for the color theme and adds the bindings to the page definition file
- Adds the necessary color theme tags to the JSF page within the map XML

24.5.6.1 Binding XML for a Color Theme

[Example 24–13](#) shows the row set bindings that were generated for the color theme of the geographic map.

Example 24–13 Color Theme Binding XML

```
<mapTheme IterBinding="ProductPopularity1Iterator" id="ProductPopularity1"
          xmlns="http://xmlns.oracle.com/adfm/dvt">
    <mapThemeDataMap mapThemeType="color">
        <item type="location" value="StateProvince" label="StateProvince"/>
        <item type="data" value="CountAddressesStateProvince"
              label="Popularity"/>
    </mapThemeDataMap>
</mapTheme>
```

24.5.6.2 XML Code on the JSF Page for a Color Theme

[Example 24–14](#) shows the XML code generated on the JSF page for a color theme that represents product popularity in different states on the United States map.

Example 24–14 Color Theme XML Code on the JSF Page

```
<dvt:mapColorTheme id="mapColorTheme1"
                     themeName="MAP_STATES_NAME"
                     shortLabel="Product Popularity"
                     value="#{bindings.ProductPopularity1.geoMapModel}"
                     locationColumn="POLYGON_NAME">
```

```
minColor="#ff0000"
maxColor="#008200"
bucketCount="5" />
```

24.5.7 What You May Need to Know About Customizing Colors in a Map Color Theme

While you are data-binding a map color theme, you can specify only a minimum color and a maximum color for the data buckets. The map uses an algorithm to determine the colors of the buckets between the minimum and maximum. However, after the data-binding is finished, you have the option of specifying the exact color to be used for each data bucket.

In the Object Inspector, for the dvt :mapColorTheme tag you can use the colorList attribute to specify the color for each bucket. You can either bind a color array to this attribute or you can specify a string of colors using a semicolon separator.

For example, if the value of this attributes is set to: #ff0000 ; #00ff00 ; #0000ff, then the color of the first bucket is red, the second bucket is green, and the third bucket is blue.

24.5.8 How to Add a Databound Pie Graph Theme to a Geographic Map

When you create a geographic map, you can choose to create themes (point, color, and graph) in any sequence that you wish. However, only one graph theme (pie or bar) can be visible at a time on the ADF geographic map component.

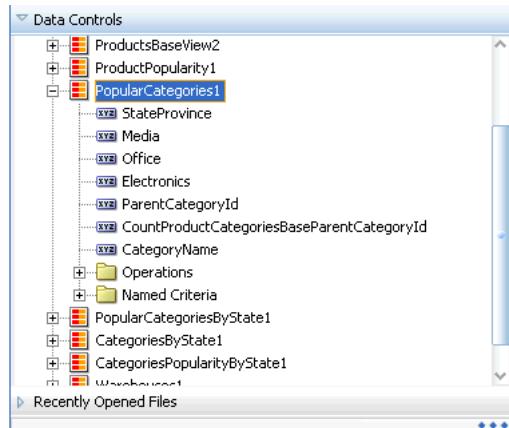
The following procedure assumes that a geographic map has already been configured and, therefore, the map component does not display the dialog for configuring the map. Instead, only the dialog for creating the pie graph theme appears.

To add a databound pie graph theme to a geographic map:

- From the Data Controls panel, select a collection.

[Figure 24–23](#) shows an example where you could select the PopularCategories1 collection to create a pie bar theme in an existing geographic map component to represent the popular product categories within a state.

Figure 24–23 Data Collection for Popular Product Categories by State

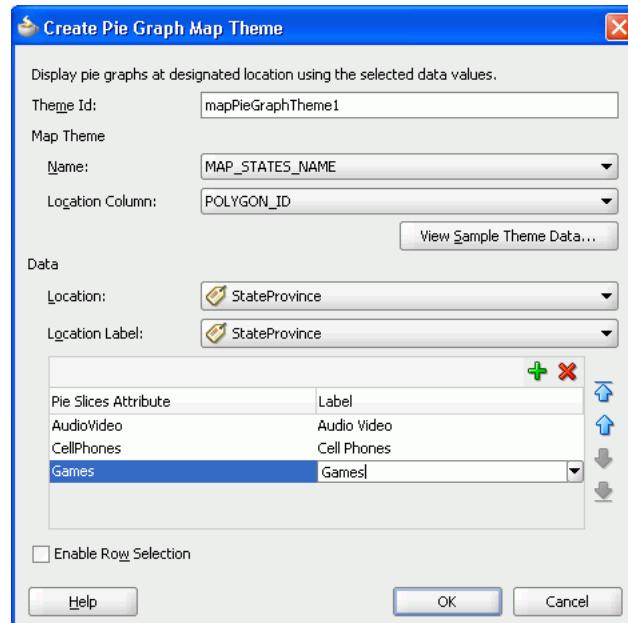


- Drag the collection onto a JSF page and, from the context menu, choose **Create**, then **Pie Graph Theme**.

3. In the ensuing Create Pie Graph Theme Binding dialog, do the following to identify the new theme and the base map theme elements that you want to work with:
 - a. For **Theme Id**, enter a unique identifier for the pie graph theme that you are creating.
 - b. In the **Base Map Theme** section, select the name of the base map and the region in which you want to place the pie graphs.
4. In the **Appearance** section, under **Data**, do the following:
 - a. For **Location**, select the location column in the data collection that should be matched to the location column in the base map theme that you selected.
If needed, click **View Sample Theme Data** to examine the first several rows of the actual data so that you can identify the appropriate location column.
 - b. For **Location Label**, select the column in the data collection that contains labels for the locations in the data collection.
 - c. In the grid for **Series Attributes**, enter each attribute that contains values that you want represented in the pie graph that you are creating.
 - d. Beside each series attribute, enter text that should be used as a label for the data values in the series attribute.
5. Select **Enable Row Selection** only if you want to enable the selection of rows in a related component. You select this component when the page contains a component that is linked to a data collection that is related to the geographic map that you are creating.
6. Click **OK**.

Figure 24–24 shows the completed Create Pie Graph Map Theme dialog for the product popularity by state pie graph theme.

Figure 24–24 Create Pie Graph Map Theme for Product Popularity by State



24.5.9 What Happens When You Add a Pie Graph Theme to a Geographic Map

Dropping a pie graph theme from the Data Controls panel to an existing geographic map has the following effect:

- Creates the bindings for the pie graph theme and adds the bindings to the page definition file
- Adds the necessary pie graph theme code to the JSF page within the map XML

24.5.9.1 Binding XML for a Pie Graph Theme

[Example 24–15](#) shows the row set bindings that were generated for the pie graph theme of the geographic map.

Example 24–15 Pie Graph Theme Binding XML

```
<mapTheme IterBinding="PopularCategoriesByState1Iterator"
          id="PopularCategoriesByState1"
          xmlns="http://xmlns.oracle.com/adfm/dvt">
  <mapThemeDataMap mapThemeType="pieChart">
    <item type="location" value="StateProvince" label="StateProvince"/>
    <item type="data" value="AudioVideo" label="Audio Video"/>
    <item type="data" value="CellPhones" label="Cell Phones"/>
    <item type="data" value="Games" label="Games"/>
  </mapThemeDataMap>
</mapTheme>
```

24.5.9.2 Code on the JSF Page for a Pie Graph Theme

[Example 24–16](#) shows the XML code generated on the JSF page for the pie graph theme of the geographic map.

Example 24–16 Pie Graph Theme Code on the JSF Page

```
<dvt:mapPieGraphTheme id="mapPieGraphTheme1"
                         themeName="MAP_STATES_NAME"
                         shortLabel="Popular Categories"
                         pieRadius="10"
                         styleName="comet"
                         value="#{bindings.PopularCategoriesByState1.geoMapModel}"
                         locationColumn="POLYGON_ID" />
```

24.6 Creating Databound Gantt Charts

A gantt is a type of bar graph (with time on the horizontal axis). It is used in planning and tracking projects to show tasks or resources in a time frame with a distinct beginning and end.

When you create a gantt, you can choose from the following types:

- **Project**

A project gantt lists tasks vertically and shows the duration of each task as a bar on a horizontal time line.

- **Resource Utilization**

A resource utilization gantt shows graphically whether resources are over or under allocated. It shows resources vertically while showing their allocation and, optionally, capacity on the horizontal time axis.

■ Scheduling

A scheduling gantt is based on manual scheduling boards and shows resources vertically with corresponding activities on the horizontal time axis. Examples of resources include people, machines, or rooms.

For information about customizing gantt charts after data-binding is completed, see the "Using ADF Gantt Chart Components" chapter in *Oracle Fusion Middleware Web User Interface Developer's Guide for Oracle Application Development Framework*.

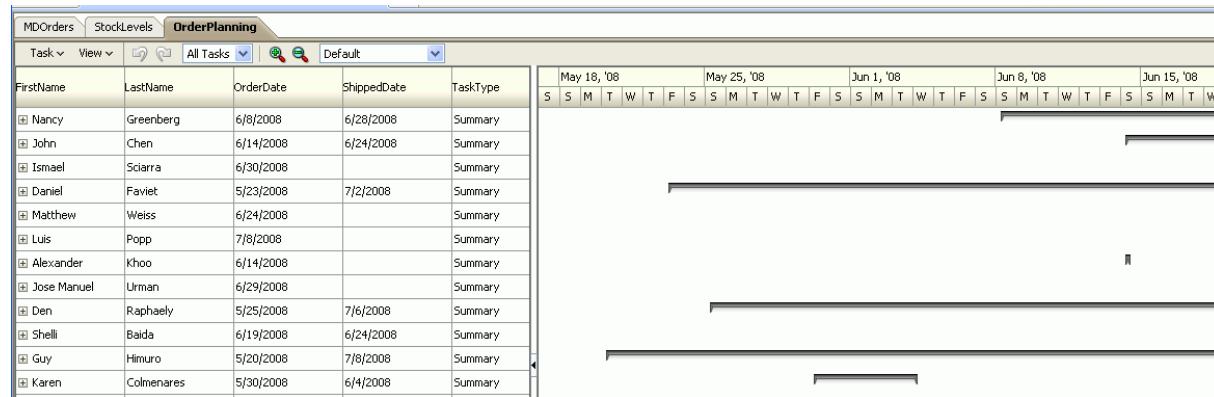
24.6.1 How to Create a Databound Project Gantt

For a project gantt, you must specify values for tasks. Optionally, you can specify values for split tasks, subtasks, recurring tasks, and dependencies between tasks, if your data collection has accessors for this additional information.

The project gantt is displayed with default values for overall start time and end time and for the major and minor time axis values. In a project gantt, the setting for the major time axis defaults to weeks and the setting for the minor time axis defaults to days.

[Figure 24–25](#) shows a project gantt in which each task is an order to be filled. The list region on the left side of the splitter shows columns with the name of the person who is responsible for the order and columns for the order date and shipped date. In the chart region on the right side of the splitter, the gantt displays a horizontal bar from the order date to the ship date for each order.

Figure 24–25 The Order Shipping Project Gantt



To create a project gantt using a data control, you bind the project gantt component to a data collection. JDeveloper allows you to do this declaratively by dragging and dropping a collection from the Data Controls panel.

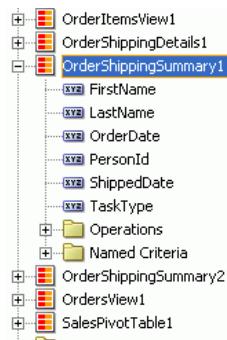
Tip: You can also create a project gantt by dragging a project gantt component from the Component Palette and completing the Create Project Gantt dialog. This approach allows you to design the gantt user interface before binding the component to data.

To create a databound project gantt:

- From the Data Controls panel, select a data collection. For a gantt, you can select a row set collection (which produces a flat list of tasks) or a basic tree collection (which produces a hierarchical list of tasks).

Figure 24–26 shows an example where you could select the OrderShippingSummary1 collection in the Data Controls panel to create a project gantt that displays the progress of order shipping.

Figure 24–26 Data Collection for Shipping Orders



2. Drag the collection onto a JSF page and, from the context menu, choose **Gantts**, then **Project**.
3. In the ensuing Create Project Gantt dialog, you do the following to connect task-related controls in the pages at the top of the dialog with corresponding columns in the data collection:
 - a. In the Tasks page at the top of the dialog, you select the columns in the data collection that correspond to each of the following controls: **Task Id**, **Start Time**, and **End Time**. Optionally, you can select a column in the data collection to map to the task type. If you do not bind **Task Type** to a column in your data collection, then all tasks default to **Normal**. The task type controls the appearance of the task bar when it is rendered in the gantt.
A project gantt component supports the following task types that you configure the data collection to return: Summary, Normal, and Milestone.
 - b. If the data collection has an accessor for subtasks, you have the option of using the Subtasks page in the dialog to select the subtasks accessor and to select the columns in the data collection that correspond to each of the following controls: **SubTask Id**, **Start Time**, and **End Time**. Optionally, you can select a column in the data collection to map to the subtask type. If you do not bind **SubTask Type** to data, then all subtasks default to **Normal**.
A project gantt component supports the following task types that you configure the data collection to return: Summary, Normal, and Milestone.

If you do not bind subtasks, then the gantt cannot render a hierarchy view of tasks. If you bind subtasks, then you can drill from tasks to subtasks in the hierarchy view of the gantt.

- c. If the data collection has an accessor for dependent tasks, you have the option of using the **Dependent Tasks** page in the dialog to select the dependent task accessor and to select the columns in the data collection that correspond to the following controls: **Dependency Type**, **From Task Id**, and **To Task Id**.

Dependent tasks are linked by their dependency between their finish and start times.

A project gantt component supports the following dependency types that you configure the data collection to return: fs (for finish to start), ss (for start to start), ff (for finish to finish), and sf (for start to finish).

- d. If the data collection has an accessor for split tasks, you have the option of using the Split Tasks page in the dialog to select **Split Tasks** accessor and to select the columns in that data collection that correspond to each of the following controls: **SplitTask Id**, **Start Time**, and **End Time**.
 - e. If the data collection has an accessor for recurring tasks, you have the option of using the Recurring Tasks page in the dialog to select the recurring tasks accessor and to select the columns in that data collection that correspond to each of the following controls: **Recurring Task Id**, **Type**, **Start Time**, and **End Time**.
4. In the **Table Columns** section, you specify the columns to appear in the list region of the gantt. Specify one row of information for each column that is to appear. Use the **New** icon to add new rows. Use the arrow icons to arrange the rows in the exact sequence that you want the columns to appear in the gantt list.

Note: The first row that you specify in the **Table Columns** section designates the nodestamp column for the list region. The nodestamp column is the one that you can expand or collapse when you have a subtask collection.

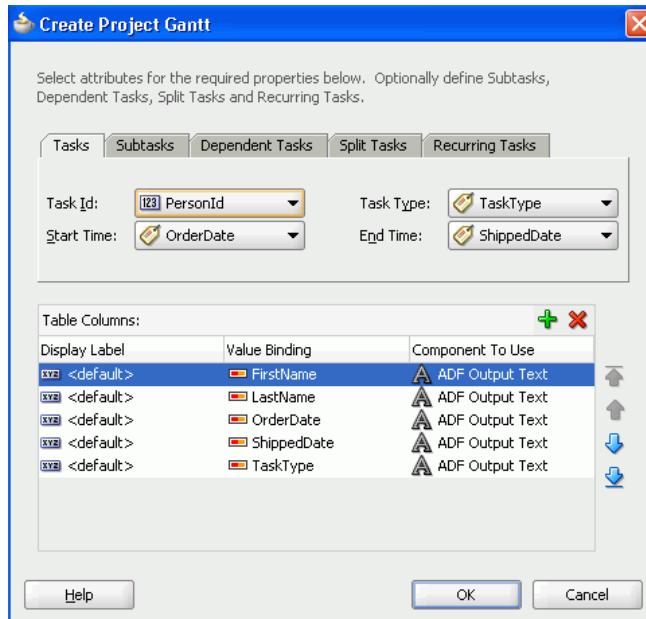
For each row, you provide the following specifications:

- **Display Label:** Select the values for the headers of the columns in the gantt list. If you select <default>, then the text for the header is automatically retrieved from the data binding.
 - **Value Binding:** Select the columns in the data collection to use for the columns in the gantt list. The available values are the same as those for the tasks group.
 - **Component to Use:** Select the type of component to display in the cell of the gantt list. The default is the ADF Output Text component.
5. Click **OK**.
6. If you want to include a legend in the gantt, right-click the project gantt node in the Structure window and choose **Insert inside dvt:projectGantt**, then **Legend**.

The legend shows information about each symbol and color coded bar that is used to represent different task types. It also shows detailed information about the task that is selected in the gantt.

Figure 24-27 shows the dialog used to create the project gantt dialog from the data collection for shipping orders.

Figure 24–27 Create Project Gantt Dialog for Orders Shipped



After completing the data binding dialog, you can use the Property Inspector to specify values for additional attributes for the project gantt.

24.6.2 What Happens When You Create a Project Gantt from a Data Control

Dropping a project gantt from the Data Controls panel has the following effect:

- Creates the bindings for the gantt and adds the bindings to the page definition file
 - Adds the necessary code for the UI components to the JSF page

Example 24-17 displays the row set bindings that were generated for the project gantt that displays orders shipped. This code example shows that there are nodes defined for tasks and subtasks, along with attributes. There are also nodes defined for dependent tasks, split tasks, and recurring tasks but no attributes.

Example 24–17 Bindings for a Project Gantt

```
<gantt IterBinding="OrderShippingSummary2Iterator"
       id="OrderShippingSummary2" xmlns="http://xmlns.oracle.com/adfm/dvt">
<ganttDataMap>
<nodeDefinition DefName="oracle.fod.model.OrderShippingSummary"
type="Tasks">
  <AttrNames>
    <Item Value="PersonId" type="taskId"/>
    <Item Value="OrderDate" type="startTime"/>
    <Item Value="TaskType" type="taskType"/>
    <Item Value="ShippedDate" type="endTime"/>
  </AttrNames>
  <Accessors>
    <Item Value="OrderShippingDetails" type="subTasks"/>
  </Accessors>
</nodeDefinition>
<nodeDefinition type="SubTasks"
DefName="oracle.fod.model.OrderShippingDetails">
  <AttrNames>
    <Item Value="OrderId" type="taskId"/>
```

```

        <Item Value="OrderDate" type="startTime"/>
        <Item Value="TaskType" type="subTaskType"/>
        <Item Value="ShippedDate" type="endTime"/>
    </AttrNames>
</nodeDefinition>
<nodeDefinition type="Dependents">
    <AttrNames/>
</nodeDefinition>
<nodeDefinition type="SplitTasks">
    <AttrNames/>
</nodeDefinition>
<nodeDefinition type="RecurringTasks">
    <AttrNames/>
</nodeDefinition>
</ganttDataMap>
</gantt>

```

Example 24–18 shows the code generated on the JSF page for the ADF project gantt. This tag code contains settings for the overall start and end time for the project gantt. It also shows the default time axis settings for the major axis (in weeks) and the minor axis (in days). Finally, it lists the specifications for each column that appears in the list region of the gantt. For brevity, the code in the af:column elements for OrderStatusCode and ShippedDate has been omitted.

Example 24–18 Code on the JSF Page for a Project Gantt

```

<projectGantt id="projectGantt1"
               value="#{bindings.OrderShippingSummary2.projectGanttModel}"
               var="row" startTime="2008-05-17" endTime="2008-07-07"
               inlineStyle="width:100%; height:100%;">
    <f:facet name="major">
        <timeAxis scale="weeks" />
    </f:facet>
    <f:facet name="minor">
        <timeAxis scale="days" />
    </f:facet>
    <f:facet name="nodeStamp">
        <af:column sortProperty="FirstName" sortable="false"
        headerText="#{bindings.OrderShippingSummary2.hints.FirstName.label}">
            <af:outputText value="#{row.FirstName}" />
        </af:column>
    </f:facet>
        <af:column sortProperty="LastName" sortable="false"
        headerText="#{bindings.OrderShippingSummary2.hints.LastName.label}">
            <af:outputText value="#{row.LastName}" />
        </af:column>
        <af:column sortProperty="OrderDate" sortable="false"
        headerText="#{bindings.OrderShippingSummary2.hints.OrderDate.label}">
            <af:outputText value="#{row.OrderDate}" />
            <af:convertDateTime
                pattern="#{bindings.OrderShippingSummary2.hints.OrderDate.format}" />
        </af:outputText>
        </af:column>
        <af:column sortProperty="ShippedDate" sortable="false"
        headerText="#{bindings.OrderShippingSummary2.hints.ShippedDate.label}">
            <af:outputText value="#{row.ShippedDate}" />
            <af:convertDateTime
                pattern="#{bindings.OrderShippingSummary2.hints.ShippedDate.format}" />
        </af:outputText>
        </af:column>
    
```

```
<af:column sortProperty="TaskType" sortable="false"
headerText="#{bindings.OrderShippingSummary2.hints.TaskType.label}">
    <af:outputText value="#{row.TaskType}" />
</af:column>
</projectGantt>
```

24.6.3 What You May Need to Know About Summary Tasks in a Project Gantt

A summary task shows start time and end time for a group of tasks (which are usually subtasks). A summary task cannot be moved or extended. However, your application should recalculate the summary task times when the dates of any subtask changes.

To detect a change in task duration, register an event handler by specifying a method binding expression on the `DataChangeListener` attribute of the Gantt component. When an action occurs that potentially changes data in the gantt, the event fired is of type `oracle.adf.view.faces.bi.event.DataChangeEvent`. This event contains information about the data changes and is fired to the registered event listener. The listener is responsible for validating the new values and updating the underlying data model.

When the update is completed, the gantt is refreshed with either the old data (if the update failed) or with the new data. The gantt uses partial page rendering so that only the gantt and not the entire page is refreshed.

24.6.4 What You May Need to Know About Percent Complete in a Project Gantt

Percent complete can be represented as an inner bar that indicates what percentage of a task is complete. The length of the inner bar is calculated based on percent complete returned by the data object.

The data binding dialog for the project gantt does not provide a control in which you can enter the percentage complete value, but this value is required to render a percent complete. However, the gantt data object does contain a `percentComplete` attribute.

To provide the percent complete integer, you must add a new attribute mapping in the `nodeDefinition` for type `Tasks`. For example, if your data collection has a column named `PercentDone`, then the attribute mapping would appear as follows: `<Item Value="PercentDone" type="percentComplete" />`.

[Example 24–19](#) shows the percent complete attribute mapping added to the data binding code for the `nodeDefinition` of type `Tasks` in the project gantt binding displayed in [Example 24–17](#).

Example 24–19 Bindings for Project Gantt with Percent Complete

```
<nodeDefinition DefName="oracle.fod.model.OrderShippingSummary"
type="Tasks">
    <AttrNames>
        <Item Value="PersonId" type="taskId"/>
        <Item Value="OrderDate" type="startTime"/>
        <Item Value="TaskType" type="taskType"/>
        <Item Value="ShippedDate" type="endTime"/>
        <Item Value="PercentDone" type="percentComplete"/>
    </AttrNames>
```

Another attribute (`completedThrough`) exists that allows you to specify a date rather than a percentage. The gantt data object calculates the percentage complete based on the date that the `completedThrough` attribute references. For example, if your data collection has a column named `PercentDone`, then the attribute mapping would

appear as follows: <Item Value="PercentDone" type="completedThrough" />.

24.6.5 What You May Need to Know About Variance in a Project Gantt

Variance can be rendered within two horizontal bars. One bar represents the base (or original) start and end time for the task. The second bar represents the actual start and end time for the task. You enter the binding information for the base start time and end time in the data binding dialog for a project gantt.

The data binding dialog for gantt does not provide controls in which you can enter the actual start time and actual end time for the gantt, but these values are required to render variance. However, the gantt data object does contain the following attributes: `actualStart` and `actualEnd`.

To provide the actual start and actual end time, you must add two new attribute mappings in the `nodeDefinition` for type Tasks. For example, if your data collection has columns named `ActualStartDate` and `ActualEndDate`, then the attribute mappings would appear as shown in [Example 24–20](#).

Example 24–20 Attribute Mappings for Actual Start and Actual End

```
<Item Value="ActualStartDate" type="actualStart"/>
<Item Value="ActualEndDate" type="actualEnd"/>
```

[Example 24–21](#) shows the actual start and actual end attribute mappings added to the data binding code for the `nodeDefinition` of type Tasks for a project gantt.

Example 24–21 Bindings for Project Gantt with Actual Start and Actual End

```
<nodeDefinition DefName="oracle.fod.model.OrderShippingSummary"
                type="Tasks">
    <AttrNames>
        <Item Value="PersonId" type="taskId"/>
        <Item Value="OrderDate" type="startTime"/>
        <Item Value="TaskType" type="taskType"/>
        <Item Value="ShippedDate" type="endTime"/>
        <Item Value="ActualStartDate" type="actualStart"/>
        <Item Value="ActualEndDate" type="actualEnd"/>
    </AttrNames>

```

24.6.6 How to Create a Databound Resource Utilization Gantt

For a resource utilization gantt, you must supply identification for resources, identification for time, and start and end times for resource usage. Optionally, you can provide data values for subresources.

The resource utilization gantt is displayed with default values for the major and minor time axis values. In a resource utilization gantt, the setting for the major time axis defaults to weeks and the setting for the minor time axis defaults to days.

[Figure 24–28](#) shows a resource utilization gantt that lists each resource and an associated calendar that can display when the resource is in use.

Figure 24–28 Resource Utilization Gantt

ResourceId	ResourceName	ServiceRegion	T	F
r1	Henry van den Broek	Northeast		
r2	Chadwick Chow	Northeast		
r3	Karin Iancu	Northeast		
r4	Imran Mohammad	Northeast		

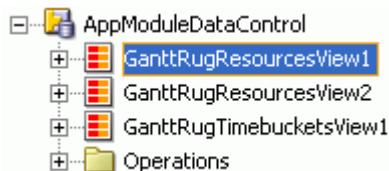
To create a resource utilization gantt using a data control, you bind the resource utilization component to a data collection. JDeveloper allows you to do this declaratively by dragging a collection from the Data Controls panel and dropping it on a JSF page.

Tip: You can also create a resource utilization gantt by dragging a resource utilization gantt component from the Component Palette and completing the Create Resource Utilization Gantt dialog. This approach gives you the option of designing the gantt user interface before binding the component to data.

To create a resource utilization gantt:

- From the Data Controls panel, select a data collection. For a gantt, you can select a row set collection or a basic tree collection.

[Figure 24–29](#) shows an example where you could select the `GanttRugResourcesView1` collection in the Data Controls panel to create a resource utilization gantt to display the usage of a resource.

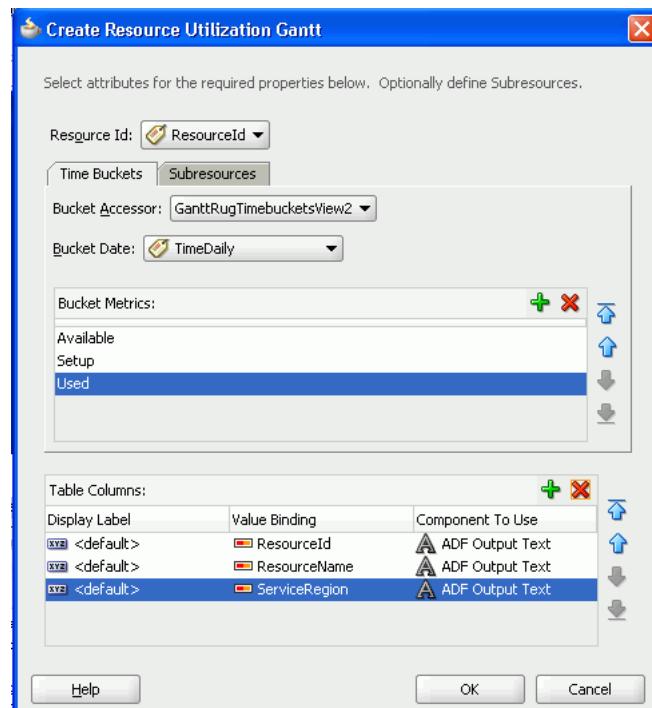
Figure 24–29 Data Collection for Resource Utilization

- Drag the collection onto a JSF page and, from the context menu, choose **Gantt**, then **Resource Utilization**.
- In the Create Resource Utilization Gantt dialog, connect resource- and time-related controls with corresponding columns in the data collection.
 - For **Resource Id**, select the column in the data collection that corresponds to the unique identifier of the resource.
 - In the **Time Buckets** page, select a value from the **Bucket Accessor** dropdown list that contains the time buckets assigned to the resource and select a value from the **Bucket Date** dropdown list that corresponds to a unit of time.

- c. In the **Bucket Metrics** list, you can optionally specify attributes that appear as bars within the time bucket. Each attribute that you specify in the **Bucket Metrics** list must be of type Number as the value of the attribute is used to calculate the height of the bar.
 - d. In the **Table Columns** list, specify the column(s) to appear in the list region of the gantt resource utilization on the left side of the splitter. Specify one row of information for each column that is to appear. Use the New icon to add new rows. Use the arrow icon to arrange the rows in the exact sequence that you want the columns to appear in the resource utilization list. For each row, provide values for **Display Label**, **Value Binding**, and **Component To Use**.
 - e. If the data collection has an accessor for subresources, you can use the **Subresources** page to select a subresources accessor and a resource ID.
4. Click **OK**.

[Figure 24–30](#) shows the dialog used to create a resource utilization gantt from the data collection for resources available for a project.

Figure 24–30 Create Resource Utilization Gantt



24.6.7 What Happens When You Create a Resource Utilization Gantt

Dropping a resource utilization gantt from the Data Controls panel onto a JSF page has the following effects:

- Creates bindings for the resource utilization gantt and adds the bindings to the page definition file
- Adds the necessary code for the UI components to the JSF page

[Example 24–22](#) shows the row set bindings that were generated for the resource utilization gantt illustrated in [Figure 24–30](#).

Example 24–22 Binding XML for a Resource Utilization Gantt

```
<gantt IterBinding="GanttRugResourcesView2Iterator"
       id="GanttRugResourcesView2"
       xmlns="http://xmlns.oracle.com/adfm/dvt">
<ganttDataMap>
<nodeDefinition DefName="model.GanttRugResourcesView" type="Resources">
    <AttrNames>
        <Item Value="ResourceId" type="resourceId"/>
    </AttrNames>
    <Accessors>
        <Item Value="GanttRugTimebucketsView2" type="timeBuckets"/>
    </Accessors>
</nodeDefinition>
<nodeDefinition type="TimeBuckets"
DefName="model.GanttRugTimebucketsView">
    <AttrNames>
        <Item Value="TimeDaily" type="time"/>
        <Item type="metric" Value="Available"/>
        <Item type="metric" Value="Setup"/>
        <Item type="metric" Value="Used"/>
    </AttrNames>
</nodeDefinition>
<nodeDefinition type="Subresources">
    <AttrNames/>
</nodeDefinition>
</ganttDataMap>
</gantt>
```

[Example 24–23](#) shows the code generated on the JSF page for the resource utilization gantt. This tag code contains settings for the overall start and end time for the resource utilization gantt. These settings have to be edited manually. The code also shows the time axis settings for the major time axis (in weeks) and the minor time axis (in days). Finally, it lists the specifications for each column to appear in the list region of the resource utilization gantt.

Example 24–23 Code on the JSF Page for a Resource Utilization Gantt

```
<dvt:resourceUtilizationGantt id="resourceUtilizationGantt1"
value="#{bindings.GanttRugResourcesView2.resourceUtilizationGanttModel}"
var="row"
metrics="#{bindings.GanttRugResourcesView2.metrics}"
taskbarFormatManager="#{bindings.GanttRugResourcesView2.resourceUtilizationGanttTa
skbarFormatManager}"
startTime="2008-07-03"
endTime="2008-07-29">
    <f:facet name="major">
        <dvt:timeAxis scale="weeks" />
    </f:facet>
    <f:facet name="minor">
        <dvt:timeAxis scale="days" />
    </f:facet>
    <f:facet name="nodeStamp">
        <af:column sortProperty="ResourceId" sortable="false"
           headerText="#{bindings.GanttRugResourcesView2.hints.ResourceId.label}">
            <af:outputText value="#{row.ResourceId}" />
        </af:column>
    </f:facet>
        <af:column sortProperty="ResourceName" sortable="false"
           headerText="#{bindings.GanttRugResourcesView2.hints.ResourceName.label}">
```

```

<af:outputText value="#{row.ResourceName}" />
</af:column>
<af:column sortProperty="ServiceRegion" sortable="false"
    headerText="#{bindings.GanttRugResourcesView2.hints.ServiceRegion.label}">
    <af:outputText value="#{row.ServiceRegion}" />
</af:column>
</dvt:resourceUtilizationGantt>

```

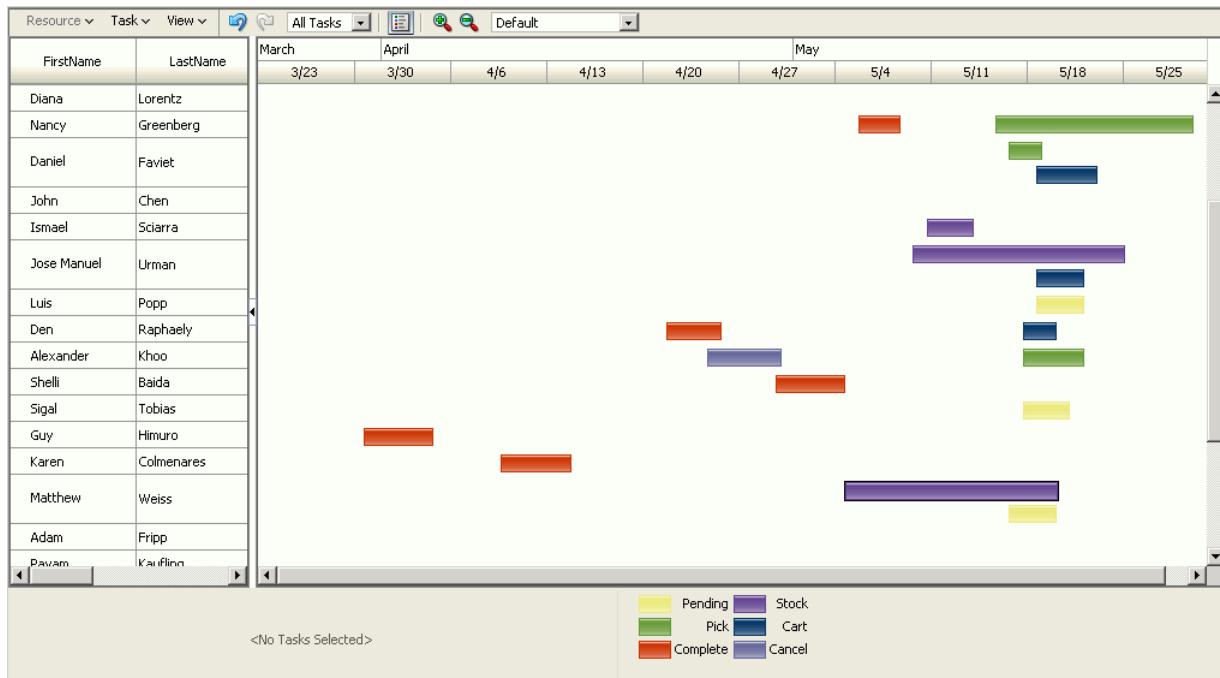
24.6.8 How to Create a Databound Scheduling Gantt

For a scheduling gantt, you must supply identification for resources, identification for tasks, and start and end times for tasks. Optionally, you can provide data values for subresources, recurring tasks, split tasks, and dependencies between tasks.

The scheduling gantt is displayed with default values for overall start and end time and for the major and minor time axis values. In a scheduling gantt, the setting for the major time axis defaults to weeks and the setting for the minor time axis defaults to days.

[Figure 24–31](#) shows a scheduling gantt that lists each resource and all the orders for which that resource is responsible. In contrast to a project gantt, the scheduling gantt shows all the tasks for a given resource on the same line, while the project gantt lists each task on a separate line.

Figure 24–31 The Scheduling Gantt for Order Shipping



To create a scheduling gantt using a data control, you bind the `schedulingGantt` tag to a data collection. JDeveloper allows you to do this declaratively by dragging and dropping a collection from the Data Controls panel.

To create a databound scheduling gantt:

- From the Data Controls panel, select a data collection. For a gantt, you can select a row set collection or a basic tree collection.

Figure 24–32 shows an example where you could select the Persons data collection to create a scheduling gantt that displays the orders that each resource is responsible for.

Figure 24–32 Data Collection for Resources



2. Drag the collection onto a JSF page and, from the context menu, choose **Gantt**, then **Scheduling**.
3. In the ensuing Create Scheduling Gantt dialog, you do the following to connect resource- and task-related controls at the top of the dialog with corresponding columns in the data collection:
 - a. For **Resource Id**, select the column in the data collection that corresponds to the unique identifier of the resource.
 - b. In the Tasks page select a value from the **Task Accessor** dropdown list that contains the tasks assigned to the resource. Select the columns in the data collection that correspond to each of the following controls: **Task Id**, **Start Time**, and **End Time**. You can optionally specify a data column to map to task type. If you do not bind task type to data, then all task types default to **Normal**.
 - c. If the data collection has an accessor that holds dependent tasks, you have the option of using the Dependent Tasks page in the dialog to select the dependent tasks accessor and to select the columns in that data collection that correspond to each of the following controls: **Dependency Type**, **From Task Id**, and **To Task Id**.
 - d. If the data collection has an accessor for split tasks, you have the option of using the Split Tasks page in the dialog to select the split tasks accessor and to select the columns in that data collection that correspond to each of the following controls: **Split Task Id**, **Start Time**, and **End Time**.
 - e. If the data collection has an accessor for recurring tasks, you have the option of using the Recurring Tasks page in the dialog to select the Recurring Tasks accessor and to select the columns in that data collection that correspond to

each of the following controls: **Recurring Task Id**, **Type**, **Start Time**, and **End Time**.

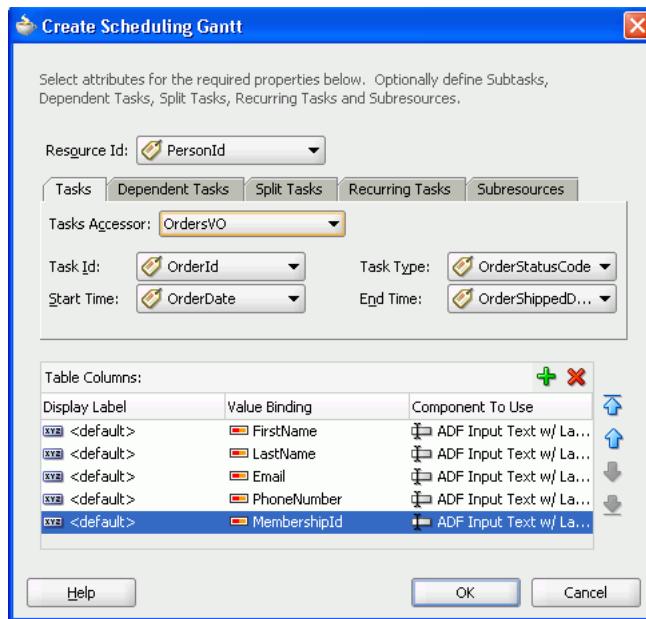
- f. If the data collection has an accessor for subresources (lower-level resources), you have the option of using the Subresources page to specify the appropriate accessor and to select the data column that contains the unique identifier of the subresource.

For example, a manager might be a resource and his direct reports might be subresources. If data contains subresources, then you can drill in a resource to locate subresources.

4. In the **Table Columns** section, you specify the columns that will appear in the list region of the gantt on the left side of the splitter. Specify one row of information for each column that is to appear. Use the **New** icon to add new rows. Use the arrow icon to arrange the rows in the exact sequence that you want the columns to appear in the gantt list. For each row, you provide the following specifications:
 - **Display Label:** Select the values for the headers of the columns in the gantt list. If you select <default>, then the text for the header is automatically retrieved from the data binding.
 - **Value Binding:** Select the columns in the data collection to use for the column in the gantt list. The available values are the same as those for the tasks group.
 - **Component to Use:** Select the type of component to display in the cell of the gantt list. The default is the **ADF Output Text** component.
5. Click **OK** to dismiss the Create Scheduling Gantt dialog.
6. Select the `dvt:schedulingGantt` element in the Structure window of the JSF page and set dates for the following attributes of the `dvt:schedulingGantt` element in the Property Inspector:
 - `StartTime`
 - `EndTime`

The dates that you specify determine the initial view that appears in the scheduling gantt at runtime.

[Figure 24–33](#) shows the dialog used to create the scheduling gantt from the data collection for resources responsible for shipping orders.

Figure 24–33 Create Scheduling Gantt Dialog

24.6.9 What Happens When You Create a Scheduling Gantt

Dropping a scheduling gantt from the Data Controls panel has the following effect:

- Creates the bindings for the gantt and adds the bindings to the page definition file
- Adds the necessary code for the UI components to the JSF page

[Example 24–24](#) shows the row set bindings that were generated for the scheduling gantt that displays resources and orders shipped.

Example 24–24 Binding XML for a Scheduling Gantt

```
<gantt IterBinding="PersonsIterator" id="Persons"
       xmlns="http://xmlns.oracle.com/adfm/dvt">
  <ganttDataMap>
    <nodeDefinition DefName="oracle.fodemo.storefront.store.queries.PersonsVO"
                    type="Resources">
      <AttrNames>
        <Item Value="PersonId" type="resourceId"/>
      </AttrNames>
      <Accessors>
        <Item Value="OrdersVO" type="tasks"/>
      </Accessors>
    </nodeDefinition>
    <nodeDefinition type="Tasks"
                   DefName="oracle.fodemo.storefront.store.queries.OrdersVO">
      <AttrNames>
        <Item Value="OrderId" type="taskId"/>
        <Item Value="OrderDate" type="startTime"/>
        <Item Value="OrderStatusCode" type="taskType"/>
        <Item Value="OrderShippedDate" type="endTime"/>
      </AttrNames>
    </nodeDefinition>
    <nodeDefinition type="Dependents">
      <AttrNames/>
    </nodeDefinition>
  </ganttDataMap>
</gantt>
```

```

<nodeDefinition type="SplitTasks">
    <AttrNames/>
</nodeDefinition>
<nodeDefinition type="RecurringTasks">
    <AttrNames/>
</nodeDefinition>
<nodeDefinition type="Subresources">
    <AttrNames/>
</nodeDefinition>
</ganttDataMap>
</gantt>

```

[Example 24–25](#) shows the code generated on the JSF page for the scheduling gantt. This tag code contains settings for the overall start and end time for the scheduling gantt. It also shows the time axis settings for the major time axis (in months) and the minor time axis (in weeks). Finally, it lists the specifications for each column that appears in the list region of the gantt. For brevity, the code in the af:column elements for MembershipId, Email, and PhoneNumber has been omitted.

Example 24–25 Code on the JSF Page for a Scheduling Gantt

```

<dvt:schedulingGantt id="schedulingGantt1"
    value="#{bindings.Persons.schedulingGanttModel}"
    var="row" startTime="2008-03-29"
    endTime="2008-05-30"
    taskbarFormatManager="#{GanttBean.taskbarFormatManager}">
    <f:facet name="major">
        <dvt:timeAxis scale="months"/>
    </f:facet>
    <f:facet name="minor">
        <dvt:timeAxis scale="weeks"/>
    </f:facet>
    <f:facet name="nodeStamp">
        <af:column sortProperty="FirstName" sortable="false"
            headerText="#{bindings.Persons.hints.FirstName.label}">
            <af:outputText value="#{row.FirstName}"/>
        </af:column>
    </f:facet>
        <af:column sortProperty="LastName" sortable="false"
            headerText="#{bindings.Persons.hints.LastName.label}">
            <af:outputText value="#{row.LastName}"/>
        </af:column>
        ...
        <dvt:ganttLegend>
    </dvt:schedulingGantt>

```


25

Creating ADF Databound Search Forms

This chapter describes how to create search forms to perform complex searches on multiple attributes and search forms to search on a single attribute. It also includes information on using named bind variables and using filtered table searches.

This chapter includes the following sections:

- [Section 25.1, "Introduction to Creating Search Forms"](#)
- [Section 25.2, "Creating Query Search Forms"](#)
- [Section 25.3, "Setting Up Search Form Properties"](#)
- [Section 25.4, "Creating Quick Query Search Forms"](#)
- [Section 25.5, "Creating Filtered Search Tables"](#)

25.1 Introduction to Creating Search Forms

You can create search forms that allow users to enter search criteria into input fields for known attributes of an object. The search criteria can be entered via input text fields or selected from a list of values in a popup list picker or dropdown list box. The entered criteria is constructed into a query to be executed. Named bind variables can be used to supply attribute values during runtime for the query. The results of the query can be displayed as a table, a form, or another UI component.

Search forms are based either on view criteria defined in view objects or implicit view criteria defined by JDeveloper. Search forms are region-based components that are reusable and personalizable. They encapsulate and automate many of the actions and iterator management operations required to perform a query. You can create several search forms on the same page without any need to change or create new iterators.

The search forms are based on the model-driven `af:query` and `af:quickQuery` components. Because these underlying components are model-driven, the search form will change automatically to reflect changes in the model. The view layer does not need to be changed. For example, if you define an LOV on an attribute in the view object or entity object, the LOV will automatically show up in the search form as an LOV component. Or, if you modify a view criteria to include a new attribute for the WHERE clause, the search panel using this view criteria will automatically reflect that change by adding a search field for that attribute.

Oracle ADF supports two types of search forms: query and quick query. The *query search form* is a full-featured search form. The *quick query search form* is a simplified form with only one search criteria. Each of these search forms can be combined with a filtered table to display the results, thereby enabling additional search capabilities. You can also create a standalone filtered table to perform searches without the query or quick query search panel.

A filtered table is a table that has additional Query-by-Example (QBE) search criteria fields above each searchable column. When the filtering option of a table is enabled, you can enter QBE-style search criteria for each column to filter the query results. For more information about tables, see [Chapter 21, "Creating ADF Databound Tables"](#).

For more information about individual query and table components, see the "Using Query Components" and the "Presenting Data in Tables and Trees" chapters of the *Oracle Fusion Middleware Web User Interface Developer's Guide for Oracle Application Development Framework*.

25.1.1 Query Search Forms

The *query search form* is the standard form for complex transactional searches. You can build complex search forms with multiple search criteria fields each with a dropdown list of built-in operators. You can also add custom operators and customize the list. It supports lists of values, **AND** and **OR** conjunctions, and saving searches for future use.

A query search form has a basic mode and an advanced mode. The user can toggle between the two modes using the basic/advanced button. At design time, you can declaratively specify form properties (such as setting the default state) to be either basic or advanced. [Figure 25–1](#) shows an advanced mode query search form with three search criteria.

Figure 25–1 Advanced Mode Query Search Form with Three Search Criteria fields

The advanced mode query form features are:

- Selecting search criteria operators from a dropdown list
- Adding custom operators and deleting standard operators
- Selecting WHERE clause conjunction of either AND or OR (match all or match any)
- Dynamically adding and removing search criteria fields at runtime
- Saving searches
- Personalizing saved searches

Typically, the query search form in either mode is used with an associated results table or tree table. For example, the query results for the search form in [Figure 25–1](#) may be displayed in a table, as shown in [Figure 25–2](#).

Figure 25–2 Results Table for a Query Search

The screenshot shows a search results table with the following items:

- 1 Treo 700w Phone/PDA**
The Palm Treo 700w smartphone delivers everything you need in a smarter phone with broadband-like speeds2 and rich-media capabilities. It has world-class ease of use to the Windows Mobile® platform. Communicate via voice, email, SMS, or MMS3. Your contacts are always ready, the web, and corporate networks on one of ...
- 2 Ipod Speakers**
The Portable Audio System for iPod and iPod Photo allows you to play your music through a speaker system. Just put your iPod in the dock, then press play. MaxxBass technology creates quality bass without a subwoofer. It provides the same synchronization, data transfer, and recharge convenience as the original iPod.
- 3 Ipod Video 80Gb**
Apple iPod - Continuing its tradition of hardware and software innovation, Apple has released a new iPod that surpasses the last. This update to the iPod's playback features a huge 80GB hard drive - the largest yet! With over 1,000 songs, there's more than enough content to satisfy your music needs. Get a 2.5" display that is now 60% brighter, a...
- 4 Ipod Shuffle 1Gb**
The Apple 1 GB Shuffle is the world's smallest digital music player. It fits in your sleeve, or your lapel, or your belt—it's so small you can carry it around with you. Put it in your iPod Shuffle and wear it as a badge of musical devotion. And because it's durable and goes with almost any outfit. About the size of a pack of gum, the iPod Shuffle measures up nicely. Just add music.

The basic mode has all the features of the advanced mode except that it does not allow the user to dynamically add search criteria fields. [Figure 25–3](#) shows a basic mode query search form with one search criteria field. Notice the lack of a dropdown list next to the **Save** button used to add search criteria fields in the advanced mode.

Figure 25–3 Basic Mode Query Form with One Search Criteria Field

The screenshot shows an 'Advanced Search' dialog box with the following configuration:

- Search Criteria:** bvProductName = equal to Ipod
- Buttons:** Search, Reset, Save...
- Labels:** Match All/Any, Find Products By Name

In either mode, each search criteria field can be modified by selecting operators such as Greater Than and Equal To from a dropdown list, and the entire search panel can be modified by the **Match All/Any** radio buttons. Partial page rendering is also supported by the search forms in almost all situations. For example, if a Between operator is chosen, another input field will be displayed to allow the user to select the upper range.

A Match All selection implicitly uses AND conjunctions between the search criteria in the WHERE clause of the query. A Match Any selection implicitly uses OR conjunctions in the WHERE clause. [Example 25–1](#) shows how the WHERE clause may appear when Match All is selected for the search criteria shown in [Figure 25–1](#).

Example 25–1 WHERE Clause When "Match All" Is Selected

```
WHERE (ProductId=4) AND (InStock > 2) AND (ProductName="Ipod")
```

[Example 25–2](#) shows how the WHERE clause may appear if **Match Any** is selected for the search criteria shown in [Figure 25–3](#).

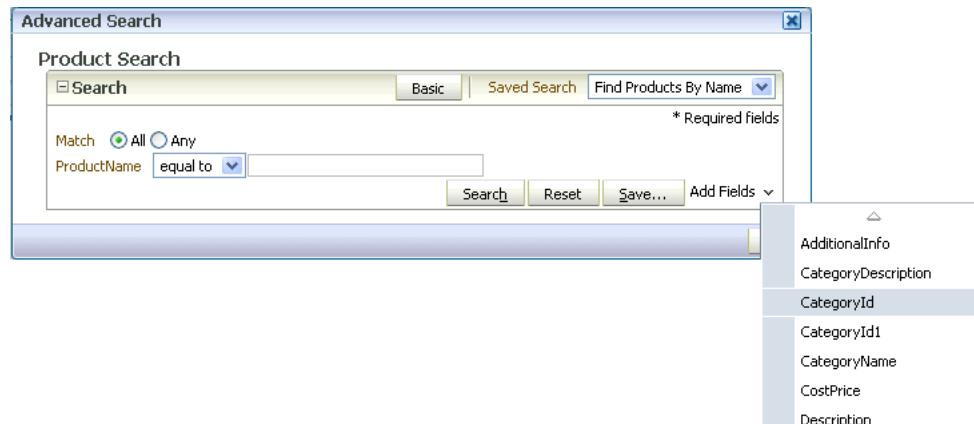
Example 25–2 WHERE Clause When Match Any Is selected

```
WHERE (ProductId=4) OR (InStock > 2) OR (ProductName="Ipod")
```

If the view criteria for the query has mixed AND and OR conjunctions between the criteria items, then neither **Match All** nor **Match Any** will be selected when the component first renders. However, the user can select **Match All** or **Match Any** to override the conjunctions defined as the initial state in the view criteria.

Advanced mode query forms allow users to dynamically add search criteria fields to the query panel to perform more complicated queries. These user-created search criteria fields can be deleted, but the user cannot delete existing fields. [Figure 25–4](#) shows how the **Add Fields** dropdown list is used to add the **CategoryId** criteria field to the search form.

Figure 25–4 Dynamically Adding Search Criteria Fields at Runtime



[Figure 25–5](#) show a user-added search criteria with the delete icon to its right. Users can click the delete icon to remove the criteria.

Figure 25–5 User-Added Search Criteria with Delete icon



If you intend for a query search form to have both a basic and an advanced mode, you can define each search criteria field to appear only for basic, only for advanced, or for both. When the user switches from one mode to the other, only the search criteria fields defined for that mode will appear. For example, three search fields for basic mode (A, B, C), and three search fields for advanced mode (A, B, D) were defined for a query. When the query search form is in basic mode, search criteria fields A, B, and C will appear. When it is in advanced mode, then field A, B, and D will appear. Any search data that was entered into the search fields will also be preserved when the form returns to that mode. In the same example, if the user entered 35 into search field

C in basic mode, then switched to advanced mode and switched back to basic, field C will reappear with value 35.

Along with using the basic or advanced mode, you can also determine how much of the search form will display. The default setting displays the whole form. You can also configure the query component to display in compact mode or simple mode. The compact mode has no header or border, and the **Saved Search** dropdown lists moves next to the expand/collapse icon. [Figure 25–6](#) shows a query component set to compact mode.

Figure 25–6 Query Component in Compact Mode

Employees (header)
This search panel can be used to search for Employees.
Saved Search System Search 1

Match All Any
*** Employee Name**:
*** Department Number**: Equals
Hire Date: Less Than

Basic | Search | Reset | Save... | Add Fields

The simple mode displays the component without the header and footer, and without the buttons normally displayed in those areas. [Figure 25–7](#) shows the same query component set to simple mode.

Figure 25–7 Query Component in Simple Mode

Search * Indicates Required Fields

Match All Any
*** Employee Name**:
**** Department Number**: Equals
Hire Date: Less Than

A query is associated with the view object that it uses for its query operation. In particular, a query component is the visual representation of the view criteria defined for that view object. If there are multiple view criteria defined, each can be selected from the **Saved Search** dropdown list. These saved searches are created at design time and can be called *system searches*. For example, in the StoreFront module of Fusion Order Demo application, there are two view criteria defined on the ProductsVO view object. When the query associated with that view criteria is run, both view criteria are available for selection, as shown in [Figure 25–8](#).

Figure 25–8 Query Form Saved Search Dropdown List

Saved Search Find Products By Name

Find Products By Name
Find Product by ID
Personalize...

Search | Reset | Save... | Add Fields

If there are no explicitly defined view criteria for a view object, you can use the default implicit view criteria.

Note: If you are developing view criteria that will be the basis for query search forms, do not use nested groups in the view criteria's expression. A query search form will support only the first group. Nested groups will not produce a runtime error, but their criteria items will be ignored by the search form.

Users can also create saved searches at runtime to save the state of a search for future use. The entered search criteria values, the basic/advanced mode state, and the layout of the results table/component can be saved by clicking the **Save** button to open a Save Search dialog. User-created saved searches persist for the session.

Figure 25–9 Runtime Saved Search Dialog Window



In addition, site administrators may create saved searches at runtime using the **Save** button. [Table 25–1](#) lists the possible scenarios for creators of saved searches, the method of their creation, and their availability.

Table 25–1 Design Time and Runtime Saved Searches

Creator	Created at Design time as View Criteria	Created at Runtime with the Save Button
Developer	Developer-created saved searches (system searches) are created during application development and typically are a part of the software release. They are created at design time as view criteria. They are usually available to all users of the application and appear in the lower part of the Saved Search dropdown list.	
Administrator		Administrator-created saved searches are created during predeployment by site administrators. They are created before the site is made available to the general end users. Administrators can create saved searches (or view criteria) using JDeveloper design time when they are logged in with the appropriate role. These saved searches (or view criteria) will appear in the lower part of the Saved Search dropdown list.

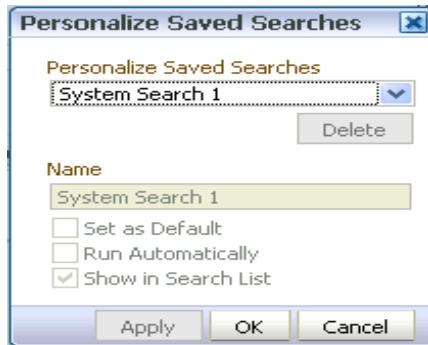
Table 25–1 (Cont.) Design Time and Runtime Saved Searches

Creator	Created at Design time as View Criteria	Created at Runtime with the Save Button
End User		End user saved searches are created at runtime using the query form Save button. They are available only to the user who created them. End user saved searches will appear in the top part of the Saved Search dropdown list.

End users can manage their saved searches by using the Personalize function in the **Saved Search** dropdown list to bring up the Personalize Saved Searches dialog, as shown in [Figure 25–10](#).

End users can use the Personalize function to:

- Update a user-created saved search
- Delete a user-created saved search
- Set a saved search as the default
- Set a saved search to run automatically
- Set the saved search to show or hide from the **Saved Search** dropdown list

Figure 25–10 Personalize Saved Searches Dialog

25.1.2 Quick Query Search Forms

A quick query search form is intended to be used in situations where a single search will suffice or as a starting point to evolve into a full query search. Both the query and quick query search forms are ADF Faces components. A quick query search form has one search criteria field with a dropdown list of the available searchable attributes from the associated data collection. Typically, the searchable attributes are all the attributes in the associated view object. You can exclude attributes by setting the attribute's **Display Hint** property in the Control Hints page of the Edit Attribute dialog to **Hide**. The user can search against the selected attribute or search against all the displayed attributes. The search criteria field type will automatically match the type of its corresponding attribute. An **Advanced** link built into the form offers you the option to create a managed bean to control switching from quick query to advanced mode query search form. For more information, see the "Using Query

Components" chapter in the *Oracle Fusion Middleware Web User Interface Developer's Guide for Oracle Application Development Framework*.

You can configure the form to have a horizontal layout, as shown in [Figure 25–11](#).

Figure 25–11 Quick Query Search Form in Horizontal Layout



You can also choose a vertical layout, as shown in [Figure 25–12](#).

Figure 25–12 Quick Query Search Form in Vertical Layout

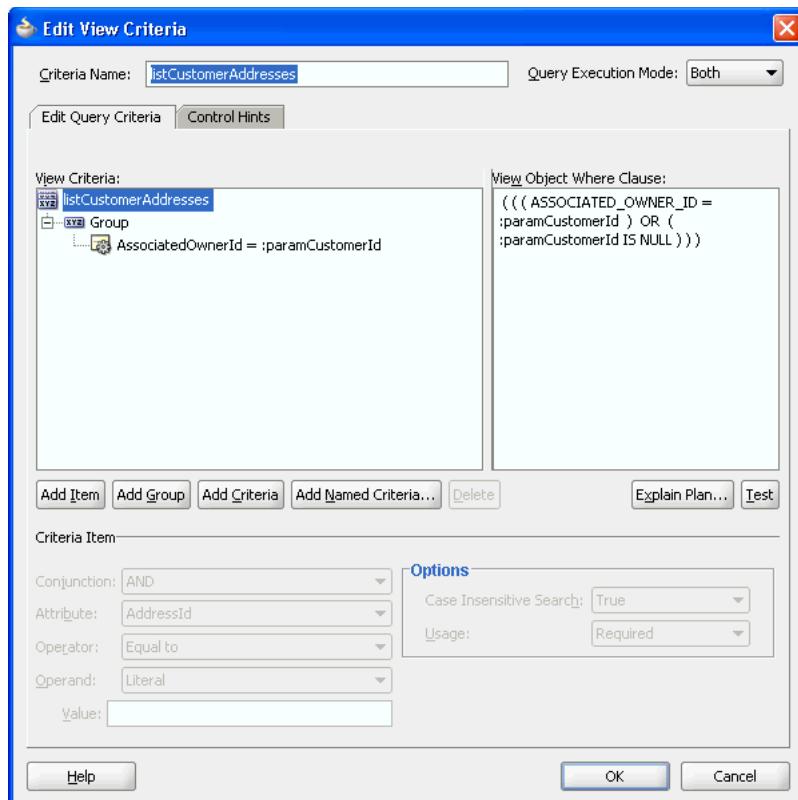


25.1.3 Named Bind Variables in Query Search Forms

Named bind variables have many different uses. For example, they can be used to pass parameter values from one page to another page, such as passing the customer ID to another page for more detail processing. And, depending on the construct of the SQL statement, using the named bind variable may speed up the query because it may lessen the need to prepare a new statement, which means that the database does not need to reparse. For more information about creating and using named bind variables, see [Section 5.9, "Working with Bind Variables"](#).

If named bind variables are used in the query defined in the view criteria, they will appear in the search form as an input field. The user can enter values for the named bind variable as in any other search criteria. At runtime, when the search form renders, the named bind variable input field may be NULL, it may contain the default value, or it may contain a value that has loaded from previous processing, such as from another page. When the search is executed, the value of the named bind variable will be evaluated with the other criteria, as defined by the SQL query statement.

For example, in the StoreFront module of the Fusion Order Demo application, in the `listCustomerAddresses` view criteria, the WHERE clause checks to see whether the `AssociatedOwnerId` is the same as the value of the `paramCustomerId` named bind variable. This view criteria is in the `AddressesLookupVO` view object. The `listCustomerAddresses` view criteria as defined in the Edit View Criteria dialog is shown in [Figure 25–13](#).

Figure 25–13 View Criteria with Named Bind Variable

A query search form for a view criteria with a named bind variable will render with a search field for the variable using the `inputText` component, as shown in [Figure 25–14](#).

Figure 25–14 Query Search Form with Named Bind Variable

25.1.4 Filtered Table and Query-by-Example Searches

A filtered table can be created standalone or as the results table of a query or quick query search form. Filtered table searches are based on Query-by-Example and use the QBE text or date input field formats. The input validators are turned off to allow for entering characters such as > and <= to modify the search criteria. For example, you can enter >1500 as the search criteria for a number column. Wildcard characters may also be supported. If a column does not support QBE, the search criteria input field will not render for that column.

The filtered table search criteria input values are used to build the query WHERE clause with the AND operator. If the filtered table is associated with a query or quick query search panel, the composite search criteria values are also combined to create the WHERE clause.

Note: If the filtered table is used with a query component in a search form and the search region is using an existing named criteria, the results of the query will be filtered by all the ViewCriteriaRow in an iterative manner. For example, a filtered table has two ViewCriteriaRow: PersonId and DeptId. The first ViewCriteriaRow has PersonId > 1. When the user enters > 100 in the filter field, then the second ViewCriteriaRow DeptId is used to accept input to further filter the results. This process iterates through all the ViewCriteriaRow until the final query result is reached.

Figure 25–15 shows a query search form with a filtered results table. When the user enters a QBE search criteria, such as >100 for the PersonId field, the query result is the AND of the query search criteria and the filtered table search criteria.

Figure 25–15 Query Search Form with Filtered Table



The screenshot shows a search interface with the following components:

- Search Buttons:** Search, Reset, Save...
- Filter Criteria:** PersonId greater than 1
- Table Headers:** PersonId, FirstName, LastName
- Table Data:**

PersonId	FirstName	LastName
110	John	Chen
126	Irene	Mikkilineni
111	Ismael	Sciarra
112	Jose Manuel	Urman
127	James	Landry
113	Luis	Popp
114	Den	Raphaely
115	Alexander	Khoo
124	Kevin	Mourgos
116	Shelli	Baida
128	Steven	Markle
117	Sigal	Tobias
118	Guy	Himuro
119	Karen	Colmenares
120	Matthew	Weiss

Table 25–2 lists the acceptable QBE search operators that can be used to modify the search value.

Table 25–2 Query-by-Example Search Criteria Operators

Operator	Description
>	Greater than
<	Less than
>=	Greater than or equal to
<=	Less than or equal to
AND	And
OR	Or

25.1.5 Implicit and Named View Criteria

During data control creation, all data collections will automatically include a Named Criteria node with an **All Queriable Attributes** criteria. This is the default view criteria that includes all the searchable attributes or columns of the data collection. You cannot edit or modify this view criteria. The implicit view criteria can be used in the same way as declaratively created view criteria during the creation of query and quick query search forms. For more information about creating named view criteria, see [Section 5.10, "Working with Named View Criteria"](#).

When you add additional view criteria for that view object or collection, the new view criteria will be added to the Named Criteria node.

Note: Query search forms do not support view criteria defined by nested expressions (multiple groups of view criteria items). However, JDeveloper allows you to drop view criteria with expressions of any kind as query search forms. For this reason, avoid selecting a view criteria with multiple groups in its expression when creating a query search form.

In the Data Controls panel, a data collection's Named Criteria node will always include the implicit view criteria, regardless of whether any named view criteria were defined. The implicit view criteria is always available for every data collection.

25.1.6 List of Values (LOV) Input Fields

List of Values (LOV) components are input components that allow the user to enter values by picking from a list that is generated by a query. ADF Faces provides the `af:inputListOfValues` and `af:inputComboboxListOfValues` components. The `af:inputListOfValues` component has a search icon next to the search criteria field, as shown in [Figure 25–16](#).

Figure 25–16 *Search Criteria Input Field Defined as an `inputListOfValues`*

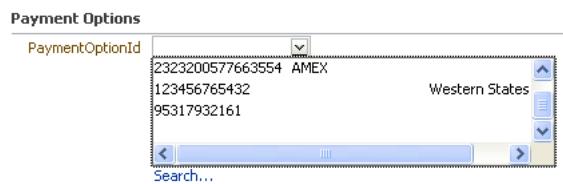


The `af:inputComboboxListOfValues` component has a dropdown icon next to the field, as shown in [Figure 25–17](#) for the `PaymentOptionId` attribute.

Figure 25–17 *Search Criteria Input Field Defined as an `inputComboboxListOfValues`*



For `af:inputComboboxListOfValues`, clicking the dropdown icon displays the LOV selection box and the **Search** command link, as shown in [Figure 25–18](#).

Figure 25–18 LOV Dropdown List with Search Link for inputComboboxListOfValues

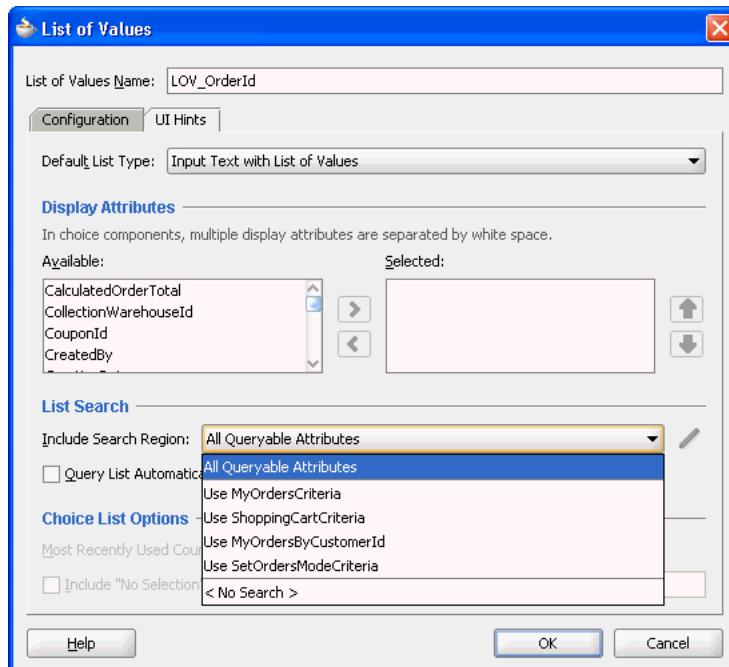
The user can select any of the items in the list to populate the input field.

The user can also click the **Search** link (or for af:inputListOfValues, the search icon) to display the LOV dialog with the full list of values in a table format, as shown in [Figure 25–19](#).

Figure 25–19 LOV Popup Search List Dialog

The LOV dialog may include a query panel to allow the user to enter criteria to search for the list of values, or it may contain only a table of results. When you define the LOV in the view object, you can specify whether a search region should be displayed and which view criteria should be used as the search to populate the list of values.

[Figure 25–20](#) shows the List of Values dialog and some of its options. In this example, the search region will be made available with **All Queriable Attributes** used for the query. For more information on LOV UI hints, see [Section 5.11.3, "How to Set User Interface Hints on a View Object LOV-Enabled Attribute"](#)

Figure 25–20 List of Values Dialog UI Hints Tab

The query and quick query components will render a search criteria field according to the **Default List Type** control hint for the corresponding attribute in the view object. Table 25–3 shows the component that will be rendered depending on the control hint.

Table 25–3 Query and Quick Query Search Criteria Field Input Components

Default List Type Control Hint	Rendered Component
Input Text with List of Values	af:inputListOfValues
Combo Box with List of Values	af:inputComboboxListOfValues
Choice List, Combo Box, List Box, Radio Group	af:selectOneChoice

View accessors must also be created to be a data source for the LOV. For more information about how to set up LOVs, see [Section 5.11, "Working with List of Values \(LOV\) in View Object Attributes"](#).

25.2 Creating Query Search Forms

You create a query search form by dropping a named view criteria item from the Data Controls panel onto a page. You have a choice of dropping only a search panel, dropping a search panel with a results table, or dropping a search panel with a tree table.

If you choose to drop the search panel with a table, you can select the filtering option in the dialog to turn the table into a filtered table.

Typically, you would drop a query search panel with the results table or tree table. JDeveloper will automatically create and associate a results table or tree table with the query panel.

If you drop a query panel by itself and want a results component or if you already have an existing component for displaying the results, you will need to match the query panel's `ResultsComponentId` with the results component's `Id`.

Note: When you drop a named view criteria onto a page, that view criteria will be the basis for the initial search form. All other view criteria defined against that data collection will also appear in the **Saved Search** dropdown list. Users can then select any of the view criteria search forms, and also any end-user created saved searches.

25.2.1 How to Create a Query Search Form with a Results Table or Tree Table

Before you begin, you should have created a view object to be the basis of the search form. If you intend to create searches using explicitly created view criteria, you need to create them, if you have not already. You can also use the default implicit view criteria, which would include all queriable attributes in the collection.

During view criteria creation, you should also set some of the search form default properties. For more information about setting the default state of the search form, see [Section 25.3.1, "How to Set Search Form Properties on the View Criteria"](#). For information on how to create view criteria, see [Section 5.10, "Working with Named View Criteria"](#).

If you intend for some or all of your input fields to be LOVs, you should declare those attributes as LOVs in the view object.

To create a query search form with a results table or tree table:

1. From the Data Controls panel, select the data collection and expand the **Named Criteria** node to display a list of named view criteria.
2. Drag the named view criteria item and drop it onto the page or onto the Structure window.
3. From the context menu, choose **Create > Query > ADF Query Panel with Table** or **Create > Query > ADF Query Panel with Tree Table**, as shown in [Figure 25–21](#).
4. In the Edit Table Columns dialog, you can rearrange any column and select table options. If you choose the filtering option, the table will be a filtered table.

After you have created the form, you may want to set some of its properties or add custom functions. For more information, see [Section 25.3, "Setting Up Search Form Properties"](#).

Figure 25–21 Data Controls Panel with Query Context Menu



25.2.2 How to Create a Query Search Form and Add a Results Component Later

Before you begin, you should have created a view object to be the basis of the search form. If you intend to create searches using your own view criteria, you need to create them, if you have not already. You can also use the default implicit view criteria, which would include all queriable attributes in the collection.

During view criteria creation, you should also set some of the search form default properties. For more information about setting the default state of the search form, see [Section 25.3.1, "How to Set Search Form Properties on the View Criteria"](#). For information on how to create view criteria, see [Section 5.10, "Working with Named View Criteria"](#).

If you intend for some or all of your input fields to be LOVs, you should declare those attributes as LOVs in the view object.

To create a query search form and add a results component in a separate step:

1. From the Data Controls panel, select the data collection and expand the **Named Criteria** node to display a list of named view criteria.
2. Drag the named view criteria item and drop it onto the page or onto the Structure window.
3. From the context menu, choose **Create > Query > ADF Query Panel**, as shown in [Figure 25–21](#).
4. If you do not already have a results component, then drop the data collection associated with the view criteria as a component.
5. In the Property Inspector for the table, copy the value of the **Id** field.
6. In the Property Inspector for the query panel, paste the value of the table's ID into the query's **ResultsComponentId** field.

After you have created the search form, you may want to set some of its properties or add custom functions. See [Section 25.3, "Setting Up Search Form Properties"](#) for more information.

25.2.3 How to Track Initial Query Execution

When a query is initially rendered within a region or a page, the query may execute. There are times when you do not want the query to execute during the initial load of the query, that is, to prevent a blind query. In this case, you want the results table to display no rows.

When a query is added, a `searchRegion` component is added to the page definition file. In the region's iterator binding, you can set the `RefreshCondition` property to an EL expression that uses the `queryPerformed` method to evaluate whether the query should be refreshed. For example:

```
RefreshCondition="#{bindings.EmpWithDeptInfoCriteriaQuery.queryPerformed}"
```

`queryPerformed` is a method on the `FacesCtrlSearchBinding` class that returns true if the query has already been executed. In this expression, the query will refresh if `queryPerformed` returns true.

[Example 25–3](#) shows how `RefreshCondition` and `queryPerformed` are used in a page definition file.

Example 25–3 Page Definition Entries for Query Search Region

```
<executables>
    <iterator Binds="AllEmployeesWithDeptInfo"
              RangeSize="25"
              DataControl="TestModuleDataControl"
              RefreshCondition="#{bindings.EmpWithDeptInfoCriteriaQuery.queryPerformed}"
              id="AllEmployeesWithDeptInfoIterator"/>
    <searchRegion Criteria="EmpWithDeptInfoCriteria"
```

```
Customizer="oracle.jbo.uicli.binding.JUSearchBindingCustomizer"
Binds="AllEmployeesWithDeptInfoIterator"
TrackingQueryPerformed="Page"
id="EmpWithDeptInfoCriteriaQuery" />
</executables>
```

You can set how long the value of queryPerformed is valid using the TrackingQueryPerformed property in the searchRegion. TrackingQueryPerformed indicates the duration for which the FacesCtrlSearchBinding should remember whether a query has been performed. When an af:query and its corresponding searchRegion binding are used in a region, you should set the value of queryPerformed to Page. When set to Page, queryPerformed will stay valid only for viewScope. If TrackingQueryPerformed is set to PageFlow, then queryPerformed is valid for pageFlowScope. The isQueryPerformed() method on FacesCtrlSearchBinding uses TrackingQueryPerformed to check for the appropriate scope and to return a boolean true or false.

25.2.4 What Happens When You Create a Query Form

When you drop a query search form onto a page, JDeveloper creates an af:query tag on the page. If you drop a query with table or tree table, then an af:table tag or af:treeTable tag will follow the af:query tag.

Under the af:query tag are several attributes that define the query properties. They include:

- The id attribute, which uniquely identifies the query.
- The resultsComponentId attribute, which identifies the component that will display the results of the query. Typically, this will be the table or tree table that was dropped onto the page together with the query. You can change this value to be the id of a different results component. For more information, see [Section 25.2.2, "How to Create a Query Search Form and Add a Results Component Later".](#)

25.2.5 What Happens at Runtime: Creating a Search Form

At runtime, the search form displays as a search panel on the page. The search panel will display in either basic mode or advanced mode, depending on the mode control hint when its corresponding view criteria was created. The **Saved Search** dropdown list will contain all the view criteria that are enabled (**Show in List** control hint enabled). The **Match All/Any** conjunction radio button may be enabled.

A search criteria field will be rendered for each search criteria defined in the view criteria. If the **Default List Type** control hint in the view object has been declared as an LOV or a selection list component, the search criteria field component is as shown in [Table 25–3](#).

After the user enters the search criteria and clicks **Search**, a query against the view criteria is executed and the results are displayed in the associated table, tree table, or component.

25.3 Setting Up Search Form Properties

A query search form is based on a view criteria defined in a view object. When you create the view criteria, you also specify some of the search form properties. Later on,

when you drop the named criteria onto the page to create a query component, you can specify other search form properties.

Search form properties that can be set when the view criteria is being created include:

- Default mode in basic or advanced mode
- Automatic query execution when the page loads
- Rendering of the search criteria field

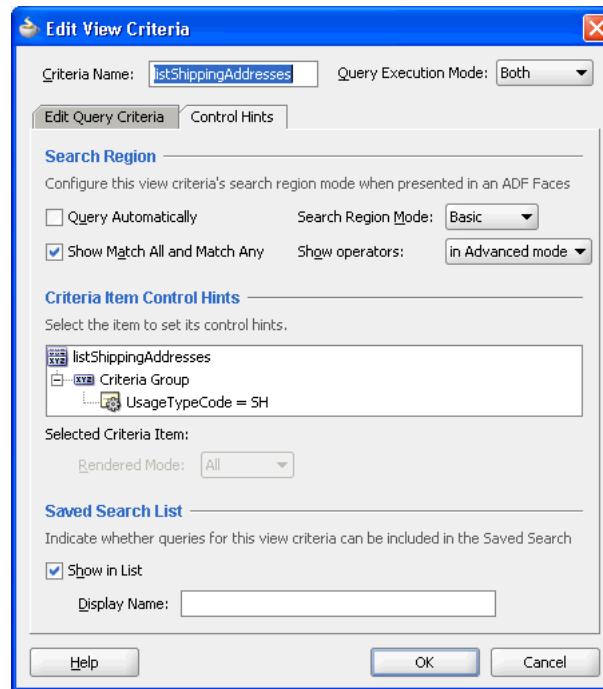
Search form properties that can be set after the query component has been added to the JSF page include:

- id of the results table or results component
- Show or hide of the basic/advanced button
- Position of the mode button
- Default, Simple, or Compact mode for display

25.3.1 How to Set Search Form Properties on the View Criteria

When you are creating a view criteria, you can declaratively set the initial state of several properties. [Figure 25–22](#) shows the Edit View Criteria dialog for setting default options. For more information about view criteria, see [Section 5.10, "Working with Named View Criteria"](#).

Figure 25–22 Edit View Criteria Dialog



You must select the default mode of the query search form as either basic or advanced. The default is basic.

You also must declare whether each individual search criteria field will be available only in basic mode, only in advanced mode, available in both modes, or never displayed. If a search criteria field is declared only for basic mode, it will not appear when the user switches to advanced mode, and the reverse is true. If the field is

declared for all, then it will appear in all modes. The default for search criteria field rendering is all modes.

To set the default mode and search criteria field display option:

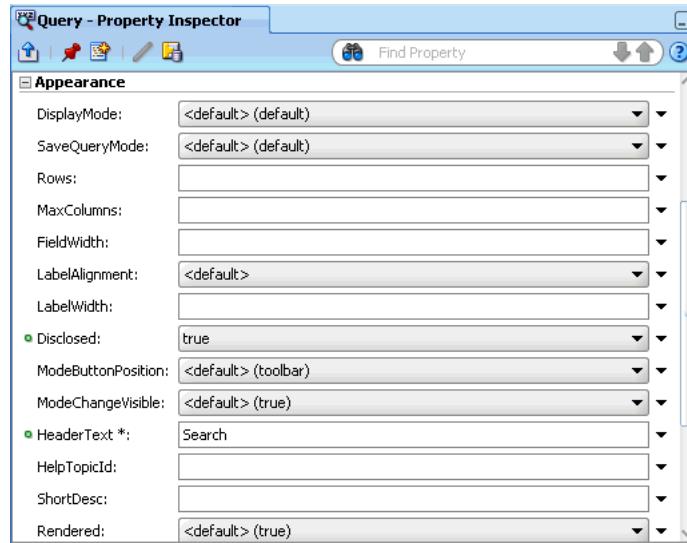
1. While creating a view criteria, in the Edit View Criteria dialog, click **Control Hints**.
2. From the **Search Region Mode** dropdown list, select either **Basic** or **Advanced**.
3. In the **Criteria Item Control Hints** section, select the criteria item you want to set.
4. In the **Rendered Mode** dropdown list, select either **All**, **Basic**, **Advanced**, or **Never**.

25.3.2 How to Set Search Form Properties on the Query Component

After you have dropped the query search form onto a page, you can edit other form properties in the Property Inspector, as shown in [Figure 25–23](#). Some of the common properties you may set are:

- Enabling or disabling the basic/advanced mode button
- Setting the ID of the query search form
- Setting the ID of the results component (for example, a results table)
- Selecting the Default, Simple, or Compact mode for display

Figure 25–23 Property Inspector for a Query Component



One common option is to show or hide the basic/advanced button.

To enable or hide the basic/advanced button in the query form:

1. In the Structure window, double-click **af:query** to open the Property Inspector.
2. Click the **Appearance** tab.
3. To enable the basic/advanced mode button, select **true** from the **ModeChangeVisible** field. To hide the basic/advance mode button, select **false** from the **ModeChangeVisible** field.

25.3.3 How to Create Custom Operators or Remove Standard Operators

You can create custom operators for each view criteria item by adding code to the view object XML file. For example, you can create a new operator called more than a year, which operates on a date attribute (greater than 365 days from the current date).

You can also remove standard operators from a view criteria item. For example, you can remove the standard operator before from the list.

For a list of standard operators, see the "Using Query Components" chapter of the *Oracle Fusion Middleware Web User Interface Developer's Guide for Oracle Application Development Framework*.

To add a custom operator:

1. In the Application Navigator, select the view object for the view criteria in which you want to add a custom operator.
2. In the editor window, click the **Source** tab to open the XML source editor.
3. Locate the code for the view criteria attribute, add the CompOper code statements after the last item within the ViewCriteriaItem group.

In [Example 25–4](#), the CompOper code statements appear in bold.

Example 25–4 Adding the Custom Operator Code to the View Object XML

```
<ViewCriteriaRow
    Name="vcrow50"
    UpperColumns="1">
    <ViewCriteriaItem
        Name="LastUpdateDate"
        ViewAttribute="LastUpdateDate"
        Operator="="
        Conjunction="AND"
        Required="Optional">
        <CompOper
            Name="LastUpdateDate"
            ToDo="1"
            OperDescStrCode="LastUpdateDate_custOp_grt_year"
            Oper=">Y"
            MinCardinality="0"
            MaxCardinality="0" >
            <TransientExpression><![CDATA[ return " < SYSDATE - 365"
                ]]></TransientExpression>
        </CompOper>
    </ViewCriteriaItem>
```

The CompOper properties are:

- Name: Specify an id for the operation.
- ToDo: Set to 1 to add this custom operator, (set to -1 to remove an operator).
- OperDescStrCode: Specify the id used in the message bundle to map to the description string, as described in Step 4.
- Oper: Set to a value that will be used programmatically to denote this operation in the SQL statement. In [Example 25–4](#), Oper was set to >Y to denote greater than 1 year.

- **MinCardinality:** If there is an input range, set this property to the minimum value for the range. For example, if the range is months in a year, this value should be set to 1. If there is no range, set it to 0.
 - **MaxCardinality:** If there is an input range, set this property to the maximum value for the range. For example, if the range is months in a year, this value should be set to 12. If there is no range, set it to 0.
 - **TransientExpression:** Set the expression to perform the custom operator function. In [Example 25–4](#), the expression is `! [CDATA[return " > SYSDATE - 365 "]]`, which returns the string " > SYSDATE - 365".
4. Open the message bundle file for the view object and add an entry for the custom operator, using the `OperDescStrCode` identifier defined in the view object XML in Step 3.

[Example 25–5](#) shows the message bundle code for the `LastUpdateDate` custom operator described in [Example 25–4](#).

Example 25–5 Adding the Custom Operator Entry for LastUpdateDate to the Message Bundle

```
public class AvailLangImplMsgBundle extends JboResourceBundle {
    static final Object[][] sMessageStrings =
    {
        { "LastUpdateDate_custOp_grt_year", "more than a year old" }
    },
```

To remove a standard operator:

1. In the Application Navigator, select the view object for the view criteria in which you want to add a custom operator.
2. In the editor window, click the **Source** tab to open the XML source editor.
3. Locate the code for the view criteria attribute, and add the `CompOper` code statements after the last item within the `ViewCriteriaItem` group.

In [Example 25–4](#), the `CompOper` code statements appears in bold. The code in this example removes the BEFORE operator from the list of operators for the `LastUpdateDate` attribute.

Example 25–6 Removing a Standard Operator Code in the View Object XML

```
<ViewCriteriaRow
    Name="vcrow50"
    UpperColumns="1">
    <ViewCriteriaItem
        Name="LastUpdateDate"
        ViewAttribute="LastUpdateDate"
        Operator="="
        Conjunction="AND"
        Required="Optional">
        <CompOper
            Name="LastUpdateDate"
            ToDo="-1"
            Oper="BEFORE"
        </CompOper>
    </ViewCriteriaItem>
```

The CompOper properties are:

- Name: Specify an id for the operation.
- ToDo: Set to -1 to remove an operator (set to 1 to add an operator).
- Oper: Set to the standard operator you want to remove from the list.

25.4 Creating Quick Query Search Forms

You can use quick query search forms to let users search on a single attribute of a collection. Quick query search form layout can be either horizontal or vertical. Because they occupy only a small area, quick query search forms can be placed in different areas of a page. You can create a managed bean to enable users to switch from a quick query to a full query search. For more information about switching from quick query to query using a managed bean, see the "Using Query Components" chapter in the *Oracle Fusion Middleware Web User Interface Developer's Guide for Oracle Application Development Framework*.

If you drop a quick query panel with a results table or tree, JDeveloper will automatically create the results table, as described in [Section 25.4.1, "How to Create a Quick Query Search Form with a Results Table or Tree Table"](#). If you drop a quick query panel by itself and subsequently want a results table or component or if you already have one, you will need to match the quick query Id with the results component's partialTrigger value, as described in [Section 25.4.2, "How to Create a Quick Query Search Form and Add a Results Component Later"](#).

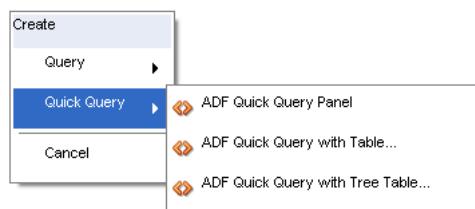
Note: A quick query search creates a dropdown list of all searchable attributes defined in the underlying view object. If you want to show only a subset of those attributes, you can set the attribute's **Display** control hint to **Hide** for those attributes you want to exclude. For more information about setting control hints on view objects, see [Chapter 5, "Defining SQL Queries Using View Objects"](#).

25.4.1 How to Create a Quick Query Search Form with a Results Table or Tree Table

You can create quick query searches using the full set of searchable attributes and simultaneously add a table or tree table as the results component.

To create a quick query search form with a results table:

1. From the Data Controls panel, select the data collection and expand the **Named Criteria** node to display a list of named view criteria.
2. Drag the **All Queriable Attributes** item and drop it onto the page or onto the Structure window.
3. From the context menu, select **Create > Quick Query > ADF Quick Query Panel with Table** or **Create > Quick Query > ADF Quick Query Panel with Tree Table**, as shown in [Figure 25-24](#).
4. In the Edit Table Columns dialog, you can rearrange any column and select table options. If you choose the filtering option, the table will be a filtered table.

Figure 25–24 Quick Query Context Menu

25.4.2 How to Create a Quick Query Search Form and Add a Results Component Later

You can create quick query searches using the full set of searchable attributes and add a table or tree table as the results component later.

To create a quick query search form and add a results component in a separate step:

1. From the Data Controls panel, select the data collection and expand the **Named Criteria** node to display a list of named view criteria.
2. Drag the **All Queriable Attributes** item and drop it onto the page or onto the Structure window.
3. From the context menu, select **Create > Quick Query > ADF Quick Query Panel**.
4. If you do not already have a results component, then drop the data collection associated with the view criteria as a component.
5. In the Property Inspector for the quick query panel, copy the value of the **Id** field.
6. In the Property Inspector for the results component (for example, a table), paste or enter the value into the **PartialTriggers** field.

25.4.3 How to Set the Quick Query Layout Format

The default layout of the form is horizontal. You can change the layout option using the Property Inspector.

To set the layout:

1. In the Structure window, double-click the **af:quickQuery** tag to open the Property Inspector.
2. On the Commons page, select the **Layout** property using the dropdown list to select either **default**, **horizontal**, or **vertical**.

25.4.4 What Happens When You Create a Quick Query Search Form

When you drop a quick query search form onto a page, JDeveloper creates an **af:quickQuery** tag. If you have dropped a quick query with table or tree table, then an **af:table** tag or **af:treeTable** tag is also added.

Under the **af:quickQuery** tag are several attributes and facets that define the quick query properties. Some of the tags are:

- The **id** attribute, which uniquely identifies the quick query. This value should be set to match the results table or component's **partialTriggers** value. JDeveloper will automatically assign these values when you drop a quick query with table or tree table. If you want to change to a different results component, see

[Section 25.4.2, "How to Create a Quick Query Search Form and Add a Results Component Later".](#)

- The `layout` attribute, which specifies the quick query layout to be either default, horizontal, or vertical.
- The `end` facet, which specifies the component to be used to display the **Advanced** link (that changes the mode from quick query to the advanced mode query). For more information about creating this function, see the "Using Query Components" chapter of the *Oracle Fusion Middleware Web User Interface Developer's Guide for Oracle Application Development Framework*.

25.4.5 What Happens at Runtime: Creating a Quick Query

At runtime, the quick query search form displays a single search criteria field with a dropdown list of selectable search criteria items. If there is only one searchable criteria item, then the dropdown list box will not be rendered. An input component that is compatible with the selected search criteria type will be displayed as shown in [Table 25–4](#). For example, if the search criteria type is date, then `inputDate` will be rendered.

Table 25–4 Quick Query Search Criteria Field Components

Attribute Type	Rendered Component
DATE	<code>af:inputDate</code>
VARCHAR	<code>af:inputText</code>
NUMBER	<code>af:inputNumberSpinBox</code>

If the **Default List Type** control hint in the view object has been declared as an LOV or a selection list component, the search criteria field component appears as shown in [Table 25–3](#) in [Section 25.1.6, "List of Values \(LOV\) Input Fields"](#).

In addition, a **Search** button is rendered to the right of the input field. If the `end` facet is specified, then any components in the `end` facet are displayed. By default, the `end` facet contains an **Advanced** link.

25.5 Creating Filtered Search Tables

You use query search forms for complex searches, but you can also perform simple QBE searches using the filtered table. You can create a standalone ADF-filtered table without the associated search panel and perform searches using the QBE-style search criteria input fields. For more information about filtered tables, see [Section 25.1.4, "Filtered Table and Query-by-Example Searches"](#).

When creating a table, you can make almost any table a filtered table by selecting the filtering option if the option is enabled. There are several ways to create a standalone filtered table. You can drop a table onto a page from the Component Palette, bind it to a data collection, and set the filtering option. For more information, see the "Using Query Components" chapter of the *Oracle Fusion Middleware Web User Interface Developer's Guide for Oracle Application Development Framework*. You can create a table by dragging and dropping a data collection onto a page and setting the filtering option. For more information, see [Section 21.2.1, "How to Create a Basic Table"](#). You can also create a filtered table by dragging and dropping named view criteria onto a page.

You can create a filtered table or a read-only filtered table by dropping named criteria onto a page. You can use either the implicitly created named criteria **All Queriable Attributes** or any declaratively created named view criteria. The resulting filtered table will have a column for each searchable attribute and a input search field above each column.

To create a filtered table using named view criteria:

1. From the Data Controls panel, expand the application module to display a list of data collections.
2. Expand the collection to display its attributes and the **Named Criteria** node.
3. Expand the **Named Criteria** node to display a list of view criteria.
4. Drag the named view criteria item and drop it onto the page or onto the Structure window.
5. From the context menu, select **Create > Tables > ADF Filtered Table** or **Create > Tables > ADF Read-Only Filtered Table**.
6. In the Edit Table Columns dialog, you can rearrange any column and select table options. Because the table is created by JDeveloper during quick query creation, the filtering option is automatically enabled and not user-selectable, as shown in [Figure 25–25](#).

Figure 25–25 Edit Table Columns Dialog for Filtered Table



26

Creating More Complex Pages

This chapter describes how to add more complex bindings to your pages. It describes how to use methods that take parameters for creating forms and command components. It also includes information about creating contextual events and using ADF model-level validation.

This chapter includes the following sections:

- [Section 26.1, "Introduction to More Complex Pages"](#)
- [Section 26.2, "Creating Command Components to Execute Methods"](#)
- [Section 26.3, "Setting Parameter Values Using a Command Component"](#)
- [Section 26.4, "Overriding Declarative Methods"](#)
- [Section 26.5, "Creating Contextual Events"](#)
- [Section 26.6, "Adding ADF Model Layer Validation"](#)
- [Section 26.7, "Displaying Error Messages"](#)
- [Section 26.8, "Customizing Error Handling"](#)

Note: Some of the implementation methods in this chapter are intended for page-level designs. If you are using task flows, you may be able to perform many of the same functions. For more information, see [Chapter 13, "Getting Started with ADF Task Flows"](#).

26.1 Introduction to More Complex Pages

Once you create a basic page and add navigation capabilities, you may want to add more complex features, such as passing parameters between pages or providing the ability to override declarative actions. Oracle ADF provides many features that allow you to add this complex functionality using very little actual code.

Some of the functions described in this chapter may be performed using other methodology. For example, if you are using task flows instead of individual page flows, you should use the task flow parameter passing mechanism. Or, if you are using Business Components, you should use the Business Components validation rules instead of ADF Model validation rules. For more information about validation rules for Business Components, see [Chapter 7, "Defining Validation and Business Rules Declaratively"](#).

26.2 Creating Command Components to Execute Methods

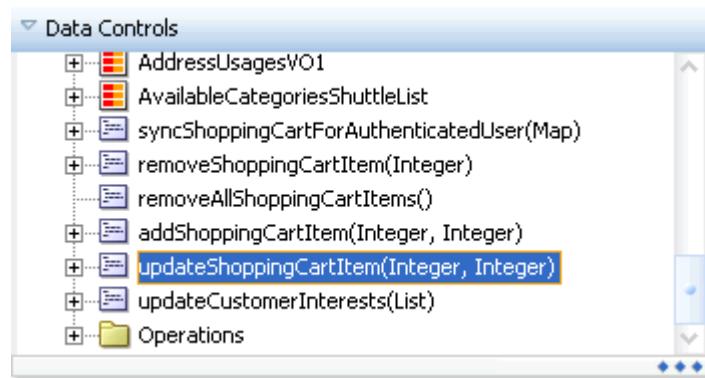
When your application contains custom methods, these methods appear in the Data Controls panel. You can then drag these methods and drop them as command buttons. When a user clicks the button, the method is executed.

For more information about creating custom methods, see [Section 9.7, "Customizing an Application Module with Service Methods"](#) and [Section 9.8, "Publishing Custom Service Methods to UI Clients"](#).

Note: If you are using task flows, you can call methods directly from the task flow definition. For more information, see [Section 14.5, "Using Method Call Activities"](#).

For example, the application module in the StoreFront module of the Fusion Order Demo application contains the `updateShoppingCartItem(Integer Integer)` method. This method updates the items in the shopping cart. To allow the user to execute this method, you drag the `updateShoppingCartItem(Integer, Integer)` method from the Data Controls panel, as shown in [Figure 26–1](#).

Figure 26–1 Methods in the Data Controls Panel



26.2.1 How to Create a Command Component Bound to a Custom Method

In order to perform the required business logic, many methods require a value for their parameter or parameters. This means that when you create a button bound to the method, you need to specify where the value for the parameter(s) is to be retrieved from.

For example, if you use the `updateShoppingCartItem(Integer Integer)` method, you need to specify the items to be updated.

To add a button bound to a method:

- From the Data Controls panel, drag the method onto the page.

Tip: If you are dropping a button for a method that needs to work with data in a table or form, that button must be dropped inside the table or form.

- From the context menu, choose **Create > Methods > ADF Button**.

If the method takes parameters, the Edit Action Binding dialog opens. In the Edit Action Binding dialog, enter values for each parameter or click the **Show EL Expression Builder** menu selection in the **Value** column of **Parameters** to launch the EL Expression Builder.

26.2.2 What Happens When You Create Command Components Using a Method

When you drop a method as a command button, JDeveloper:

- Defines a method action binding for the method. Action bindings use the `RequiresUpdateModel` property, which determines whether or not the model needs to be updated before the action is executed. For command operations, this property is set to `true` by default, which means that any changes made at the view layer must be moved to the model before the operation is executed.
- If the method takes any parameters, JDeveloper creates `NamedData` elements that hold the parameter values.
- Inserts code in the JSF page for the ADF Faces command component. This code is the same as code for any other command button, as described in [Section 20.4.2.3, "Using EL Expressions to Bind to Navigation Operations"](#). However, instead of being bound to the `execute` method of the action binding for a built-in operation, the button is bound to the `execute` method of the method action binding for the method that was dropped.

26.2.2.1 Using Parameters in a Method

When you drop a method that takes parameters onto a JSF page, JDeveloper creates a method action binding. This binding is what causes the method to be executed when a user clicks the command component. When the method requires parameters to run, JDeveloper also creates `NamedData` elements for each parameter. These elements represent the parameters of the method.

For example, the `updateShoppingCartItem(Integer, Integer)` method action binding contains `NamedData` elements for the parameter. This element is bound to the value specified when you created the action binding. [Example 26–1](#) shows the method action binding created when you drop the `updateShoppingCartItem(Integer Integer)` method, and bind the `Integer` parameter (named `productId`) and the other `Integer` parameter (named `quantity`) to the appropriate variables.

Example 26–1 Method Action Binding for a Parameter Method

```
<methodAction id="updateShoppingCartItem" RequiresUpdateModel="true"
    Action="invokeMethod" MethodName="updateShoppingCartItem"
    IsViewObjectMethod="false"
    DataControl="StoreServiceAMDataControl"
    InstanceName="StoreServiceAMDataControl.dataProvider">
    <NamedData NDName="productId" NDValue="4" NDTType="java.lang.Integer"/>
    <NamedData NDName="quantity" NDValue="5" NDTType="java.lang.Integer"/>
</methodAction>
```

26.2.2.2 Using EL Expressions to Bind to Methods

Like creating command buttons using operations, when you create a command button using a method, JDeveloper binds the button to the method using the `actionListener` attribute. The button is bound to the `execute` property of the action binding for the given method using an EL expression. This EL expression causes

the binding's method to be invoked on the application module. For more information about the command button's `actionListener` attribute, see [Section 20.4.3, "What Happens at Runtime: How Action Events and Action Listeners Work"](#).

Tip: Instead of binding a button to the `execute` method on the action binding, you can bind the button to the method in a backing bean that overrides the `execute` method. Doing so allows you to add logic before or after the original method runs. For more information, see [Section 26.4, "Overriding Declarative Methods"](#).

Like navigation operations, the `disabled` property on the button uses an EL expression to determine whether or not to display the button. [Example 26–2](#) shows the EL expression used to bind the command button to the `updateShoppingCartItem(Integer, Integer)` method.

Example 26–2 JSF Code to Bind a Command Button to a Method

```
<af:commandButton actionListener="#{bindings.updateShoppingCartItem.execute}"  
                  text="updateShoppingCartItem"  
                  disabled="#{!bindings.updateShoppingCartItem.enabled}"/>
```

Tip: When you drop a command button component onto the page, JDeveloper automatically gives it an ID based on the number of the same type of component that was previously dropped. For example, `commandButton1`, `commandButton2`. If you change the ID to something more descriptive, you must manually update any references to it in any EL expressions in the page.

26.2.2.3 Using the Return Value from a Method Call

You can also use the return value from a method call. [Example 26–3](#) shows a custom method that returns a string value.

Example 26–3 Custom Method That Returns a Value

```
/**  
 * Custom method.  
 */  
public String getHelloString() {  
    return ("Hello World");  
}
```

[Example 26–4](#) shows the code in the JSPX page for the command button and an `outputText` component.

Example 26–4 Command Button to Call the Custom Method

```
<af:commandButton actionListener="#{bindings.getHelloString.execute}"  
                  text="getHelloString"  
                  disabled="#{!bindings.getHelloString.enabled}"  
                  id="helloButtonId"/>  
<af:outputText value="#{bindings.return.inputValue}"  
                  id="helloOutputId"/>
```

When the user clicks the command button, it calls the custom method. The method returns the string "Hello World" to be shown as the value of the `outputText` component.

26.2.3 What Happens at Runtime: Binding a Method to a Command Button

When the user clicks the button, the method binding causes the associated method to be invoked, passing in the value bound to the `NamedData` element as the parameter. For example, if a user clicks a button bound to the `updateShoppingCartItem(Integer, Integer)` method, the method takes the values of the product `Id` and quantity and updates the shopping cart.

26.3 Setting Parameter Values Using a Command Component

There may be cases where an action on one page needs to set parameters that will be used to determine application functionality. For example, you can create a search command button on one page that will navigate to a results table on another page. But the results table will display only if a parameter value is `false`.

You can use a managed bean to pass this parameter between the pages, and to contain the method that is used to check the value of this parameter. The managed bean is instantiated as the search page is rendered, and a method on the bean checks that parameter. If it is `null` (which it will be the first time the page is rendered), the bean sets the value to `true`.

For more information about creating custom methods, see [Section 9.7, "Customizing an Application Module with Service Methods"](#) and [Section 9.8, "Publishing Custom Service Methods to UI Clients"](#).

Note: If you are using task flows, you can use the task flow parameter passing mechanism. For more information, see [Chapter 15, "Using Parameters in Task Flows"](#).

A `setPropertyListener` component with `type` property set to `action`, which is nested in the command button that executed this search, is then used to set this flag to `false`, thus causing the results table to display once the search is executed. For information about using managed beans, see [Section 18.4, "Using a Managed Bean in a Fusion Web Application"](#).

26.3.1 How to Set Parameters Using `setPropertyListener` Within a Command Component

You can use the `setPropertyListener` component to set values on other objects. This component must be a child of a command component.

Before you add the `setPropertyListener` to the command component, you must have already created a command component on the page.

To use the `setPropertyListener` component:

1. From the Component Palette, drag a `setPropertyListener` component and drop it as a child of the command component.
Or right-click the component and select **Insert inside Button > ADF Faces > setPropertyListener**.
2. In the Insert Set Property Listener dialog, set **From** to be the parameter value.
3. Set **To** to be where you want to set the parameter value.

Tip: Consider storing the parameter value on a managed bean or in scope instead of setting it directly on the resulting page's page definition file. By setting it directly on the next page, you lose the ability to easily change navigation in the future. For more information, see [Section 18.4, "Using a Managed Bean in a Fusion Web Application"](#). Additionally, the data in a binding container is valid only during the request in which the container was prepared. The data may change between the time you set it and the time the next page is rendered

4. Select **Action** from the **Type** dropdown menu.

26.3.2 What Happens When You Set Parameters

The `setPropertyListener` component lets the command component set a value before it navigates to the next page. When you set the `From` attribute either to the source of the value you need to pass or to the actual value, the component will be able to access that value. When you set the `To` attribute to a target, the command component is able to set the value on the target. [Example 26–5](#) shows the code on the JSF page for a command component that takes the value `false` and sets it as the value of the `initialSearch` flag on the `searchResults` managed bean.

Example 26–5 JSF Page Code for a Command Button Using a `setPropertyListener` Component

```
<af:commandButton actionListener="#{bindings.Execute.execute}"  
    text=Search>  
    <af:setPropertyListener from="#{false}"  
        to="#{searchResults.initialSearch}"/>  
        type="action"/>  
</af:commandButton>
```

26.3.3 What Happens at Runtime: Setting Parameters Using Command Component

When a user clicks the command component, before navigation occurs, the `setPropertyListener` component sets the parameter value. In [Example 26–5](#), the `setPropertyListener` takes the value `false` and sets it as the value for the `initialSearch` attribute on the `searchResults` managed bean. Now, any component that needs to know this value in determining whether or not to render can access it using the EL expression `#{searchResults.initialSearch}`.

26.4 Overriding Declarative Methods

When you drop an operation or method as a command button, JDeveloper binds the button to the `execute` method for the operation or method. However, there may be occasions when you need to add logic before or after the existing logic.

Note: If you are using task flows, you can call custom methods from the task flow. For more information, see [Chapter 13, "Getting Started with ADF Task Flows"](#).

JDeveloper allows you to add logic to a declarative operation by creating a new method and property on a managed bean that provides access to the binding container. By default, this generated code executes the operation or method. You can

then add logic before or after this code. JDeveloper automatically binds the command component to this new method, instead of to the `execute` property on the original operation or method. Now when the user clicks the button, the new method is executed.

For example, in the Fusion Order Demo application Orders page, the `Commit` operation requires additional processing. The `Commit` button is renamed to `Submit Orders` and logic is added to the `submitOrders` method in the `orderPageBean` managed bean.

26.4.1 How to Override a Declarative Method

In order to override a declarative method, you must have a managed bean to hold the new method to which the command component will be bound. If your page has a backing bean associated with it, JDeveloper adds the code needed to access the binding object to this backing bean. If your page does not have a backing bean, JDeveloper asks you to create one.

Note: You cannot override the declarative method if the command component currently has an EL expression as its value for the `Action` attribute, because JDeveloper will not overwrite an EL expression. You must remove this value before continuing.

To override a declarative method:

1. Drag the operation or method to be overridden onto the JSF page and drop it as a UI command component.

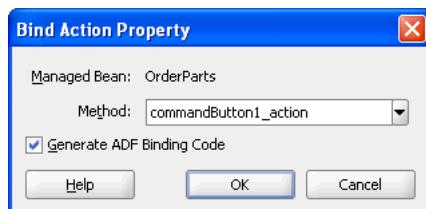
The component is created and bound to the associated binding object in the ADF Model layer with the `ActionListener` attribute.

For more information about creating command components using methods on the Data Controls panel, see [Section 26.2, "Creating Command Components to Execute Methods"](#).

For more information about creating command components from operations, see [Section 20.4.2, "What Happens When You Create Command Buttons"](#)

2. On the JSF page, double-click the component.
3. In the Bind Action Property dialog, identify the backing bean and the method to which you want to bind the component using one of the following techniques:
 - If auto-binding has been enabled on the page, the backing bean is already selected for you, as shown in [Figure 26–2](#).

Figure 26–2 Bind Action Property Dialog for a Page with Auto-Binding Enabled



- To create a new method, enter a name for the method in the **Method** field, which initially displays a default name.

or

- To use an existing method, select a method from the dropdown list in the **Method** field.
- Select **Generate ADF Binding Code**.
- If the page is not using auto-binding, you can select from an existing backing bean or create a new one, as shown in [Figure 26–3](#).

Figure 26–3 Bind Action Property Dialog for a Page with Auto-Binding Disabled



- Click **New** to create a new backing bean. In the Create Managed Bean dialog, name the bean and the class, and set the bean's scope.
- or
- Select an existing backing bean and method from the dropdown lists.

Note: Whenever there is a value for the `ActionListener` attribute on the command component, JDeveloper understands that the button is bound to the `execute` property of a binding. If you have removed that binding, you will not be given the choice to generate the ADF binding code. You will need to insert the code manually, or to set a dummy value for the `ActionListener` before double-clicking the command component.

4. After identifying the backing bean and method, click **OK** in the Bind Action Property dialog

JDeveloper opens the managed bean in the source editor. [Example 26–6](#) shows the code inserted into the bean. In this example, a command button is bound to the `Commit` operation.

Example 26–6 Generated Code in a Backing Bean to Access the Binding Object

```
public String submitOrder() {
    BindingContainer bindings = getBindings();
    OperationBinding operationBinding =
        bindings.getOperationBinding("Commit");
    Object result = operationBinding.execute();
    if (!operationBinding.getErrors().isEmpty()) {
        return null;
    }
}
```

5. You can now add logic either before or after the binding object is accessed, as shown in [Example 26–7](#).

Example 26–7 Code Added to the Overridden Method

```

public String submitOrder() {
    DCBindingContainer bindings =
        (DCBindingContainer)JSFUtils.resolveExpression("#{bindings}");
    OperationBinding operationBinding =
        bindings.getOperationBinding("Commit");
    JUCtrlAttrsBinding statusCode =
        (JUCtrlAttrsBinding)bindings.findNamedObject("OrderStatusCode");
    statusCode.setAttribute("OrderStatusCode", "PENDING");
    JUCtrlAttrsBinding orderDate =
        (JUCtrlAttrsBinding)bindings.findNamedObject("OrderDate");
    orderDate.setAttribute("OrderDate", new Date());
    JUCtrlAttrsBinding orderId =
        (JUCtrlAttrsBinding)bindings.findNamedObject("OrderId");
    System.out.println("OrderId = " + orderId);
    JSFUtils.storeOnSession("orderId", orderId.getAttribute("OrderId"));
    Object result = operationBinding.execute();
    if (!operationBinding.getErrors().isEmpty()) {
        return null;
    }
}

```

In addition to any processing logic, you may also want to write conditional logic to return one of multiple outcomes. For example, you might want to return `null` if there is an error in the processing, or another outcome value if the processing was successful. A return value of `null` causes the navigation handler to forgo evaluating navigation cases and to immediately redisplay the current page.

Tip: To trigger a specific navigation case, the outcome value returned by the method must exactly match the outcome value in the navigation rule, including case.

The command button is now bound to this new method using the `Action` attribute instead of the `ActionListener` attribute. If a value had previously existed for the `Action` attribute (such as an outcome string), that value is added as the return for the new method. If there was no value, the return is kept as `null`.

26.4.2 What Happens When You Override a Declarative Method

When you override a declarative method, JDeveloper adds a managed property to your backing bean with the managed property value of `#{bindings}` (the reference to the binding container), and it adds a strongly typed bean property to your class of the `BindingContainer` type, which the JSF runtime will then set with the value of the managed property expression `#{bindings}`. JDeveloper also adds logic to the UI command action method. This logic includes the strongly typed `getBindings()` method used to access the current binding container.

The code does the following:

- Accesses the binding container.
- Finds the binding for the associated method, and executes it.
- Adds a return for the method that can be used for navigation. By default, the return is `null`. If an outcome string had previously existed for the button's `Action` attribute, that attribute is used as the return value. You can change this code as needed.

JDeveloper automatically rebinds the UI command component to the new method using the `Action` attribute, instead of the `ActionListener` attribute. [Example 26–8](#) shows the code when a `Commit` operation is declaratively added to a page.

Example 26–8 JSF Page Code for a Command Button Bound to a Declarative Method

```
<af:commandButton actionListener="#{bindings.Commit.execute}"
                  text="Commit"
                  disabled="#{!bindings.Commit.enabled}" />
```

[Example 26–9](#) shows the code after the method on the page's backing bean is overridden. Note that the `action` attribute is now bound to the backing bean's method.

Example 26–9 JSF Page Code for a Command Button Bound to an Overridden Method

```
<af:commandButton text="Submit Order"
                  action="#{orderPageBean.submitOrder}" />
```

Tip: If when you click the button that uses the overridden method you receive this error:

SEVERE: Managed bean `main_beans` could not be created
The scope of the referenced object: '`#{bindings}`' is shorter than the referring object

it is because the managed bean that contains the overriding method has a scope that is greater than `request` (that is, either `session` or `application`). Because the data in the binding container referenced in the method has a scope of `request`, the scope of this managed bean must be set to the same or a lesser scope.

26.5 Creating Contextual Events

Often a page or a region within a page needs information from somewhere else on the page or from a different region. While you can pass parameters to obtain that information, doing so makes sense only when the parameters are well known and the inputs are EL-accessible to the page. Parameters are also useful when a task flow may need to be restarted if the parameter value changes.

However, say you have a task flow with multiple page fragments that contain various interesting values that could be used as input on one of the pages in the flow. If you were to use parameters to pass the value, the task flow would need to surface output parameters for the union of each of the interesting values on each and every fragment. Instead, for each fragment that contains the needed information, you can define a contextual event that will be raised when the page is submitted. The page or fragment that requires the information can then subscribe to the various events and receive the information through the event.

For example, in the StoreFront module, a contextual event is used on the home page to determine the products to display on the product table based on the category selected in the category tree. When a user selects a category (for example, Electronics), the `categoriesTaskFlow1` region that contains the tree raises an event, taking the selected category as its payload. An event map created in the home page's page definition file maps that event to the consumer, which is the region that contains the products table. Using the payload, a method binding on the product table region's

page definition executes to determine the correct products to display based on the selected category.

Events are configured in the page definition file for the page or region that will raise the event (the producer). In order to associate the producer with the consumer that will do something based on the event, you create an event map also in the page definition (when using events between regions, the page definition file for the page in the region that consumes the event contains the event map). Even if the consuming page is in a dynamic region, the event map should be in the page definition file of the consuming page.

You can raise a contextual event for an action binding, a method action binding, a value attribute binding, a tree binding, a table binding, or a list binding. For action and method action bindings, the event is raised when the action or method is executed. For a value attribute binding, the event is triggered by the binding container and raised after the attribute is set successfully. For a range binding (tree, table, list), the event is raised after the currency change has succeeded. Value attribute binding and range binding contextual events may also be triggered by navigational changes.

Contextual events are not the same as events raised by UI components. For a description of these types of events, see the *Oracle Fusion Middleware Web User Interface Developer's Guide for Oracle Application Development Framework*. Contextual events can be used in association with UI events. In this case, an action listener that is invoked due to a UI event can, in turn, invoke a method action binding that then raises the event.

26.5.1 How to Create Contextual Events

You create contextual events by first creating the event on the producer. You then determine the consumer of the event and map the producer and consumer.

Before you create a contextual event, you must have a method binding, action binding, value attribute binding, or list binding on the page. If you do not, you must create the binding first. For example, for a method binding, in the Structure window, right-click the binding and choose **Insert inside bindings > Generic Bindings > methodAction** and add the method action binding. For the other bindings, you may need to drop a component such as an input text, table, or tree to the page.

To create a contextual event:

1. Open the page definition file that contains the binding for the producer of the event.
A producer must have an associated binding that will be used to raise the event. For example, if a method or operation will be the producer, the associated action binding or method action binding will contain the event.
2. If you do not have a method binding, action binding, value attribute binding, or list binding on the page, you need to create the binding first. For example, for a method binding, in the Structure window, right-click the binding and choose **Insert inside bindings > Generic Bindings > methodAction** and add the method action binding. For the other bindings, you may need to drop a component such as an input text, table, or tree to the page.
3. In the Structure window, right-click the binding for the producer and choose **Insert inside binding name > events** or **Insert inside binding name > Contextual Events > events**.
4. In the Structure window, right-click the events element just created, and choose **Insert inside events > event**.

5. In the Insert event dialog, enter a name for the event in the **name** field, and click **Finish**.

The event is now created. By default, any return of the associated method or operation will be taken as the payload for the event and stored in the EL-accessible variable \${payLoad}. You now need to map the event to the consumer, and to configure any payload that needs to be passed to the consumer.

6. Open the page definition that contains the binding for the consumer.

The binding container represented by this page provides access to the events from the current scope, including all contained binding containers (such as task flow regions). If regions or other nested containers need to be aware of the event, the event map should be in the page definition of the page in the consuming region.

7. In the Structure window, right-click the topmost node that represents the page definition, and choose **Edit Event Map**.

Note: If the producer event comes from a page in an embedded dynamic region, you may not be able to edit the event map using the Event Map editor. You can manually create the event map by editing the page definition file or use **insert inside** steps as described in [Section 26.5.2, "How to Manually Create the Event Map"](#).

8. In the Edit Event Map dialog, do the following:

1. Use the **Producer** dropdown menu to choose the producer.
2. Use the **Event Name** dropdown menu to choose the event.
3. Use the **Consumer** dropdown menu to choose the consumer. This should be the actual method that will consume the event.
4. If the consuming method or operation requires parameters, click the **Parameters** ellipses button. In the Edit Consumer Params dialog, in the **Param Name** field, enter the name of the parameter expected by the method. In the **Param Value** field, enter the value. If this is to be the payload from the event, you can access this value using the \${payLoad} expression. If the payload contains many parameters and you don't need them all, use the ellipses button to open the Expression Builder dialog. You can use this dialog to select specific parameters under the **payload** node.
5. In the Event Map Editor dialog, click **Add New** to add the event map to the event maps in the page definition.
6. Click **OK** to add the map to the page definition file.

26.5.2 How to Manually Create the Event Map

Under most circumstances, you can create the event map using the Event Map Editor as described in [Section 26.5.1, "How to Create Contextual Events"](#). However, in situations such as when the producer event is from a page in an embedded dynamic region, the Event Map Editor at design time cannot obtain the necessary information to create an event map.

To create the event map manually:

1. Open the page definition that contains the binding for the consumer.

2. In the Structure window, right-click the topmost node that represents the page definition, and choose **Insert inside pagedef name > eventMap**.
An **eventMap** node appears in the Structure pane.
3. Select the **eventMap** node, right-click and choose **Insert inside eventMap > event**.
In the Insert Event dialog, enter the name of the event and click **OK**.
An **event** node appears under the **eventMap** node.
Repeat this step to add more events.
4. Select **event**, right-click and choose **Insert inside event > producer**.
The Insert producer dialog appears. Enter the name of the binding that is producing this event. You can also enter the name of the producer region, in which case all the consumers specified under this tag can consume the event. You can also enter "*" to denote that this event is available for all consumers under this tag. Click **OK**.
A **producer** node appears under the **event** node.
5. Select **producer**, right-click and choose **Insert inside producer > consumer**.
The Insert consumer dialog appears. Enter the name of the handler that will consume the event. Click **OK**.
A **consumer** node appears under the **producer** node.
Repeat this step to add more consumers.
6. If there are parameters being passed, add the parameter name and value. Select **consumer**, right-click and choose **Insert inside consumer > parameters**.
A **parameters** node appears under the **consumer** node.
Select **parameters**, right-click and choose **Insert inside parameters > parameter**.
The Insert parameter dialog appears. Enter the name of the parameter and the value of the parameter. The value can be an EL expression. Click **OK**.
Repeat **Insert inside parameters > parameter** to add more parameters

26.5.3 What Happens When You Create Contextual Events

When you create an event for the producer, JDeveloper adds an `events` element to the page definition file. Each event name is added as a child. [Example 26–10](#) shows the event on the `setSelectedProductCategory` method action binding in the `categories_categoriesTreePageDef` page definition file of the StoreFront module. This is the page definition for the region used to display the category tree.

Example 26–10 Event Definition for the Producer

```
<methodAction id="setSelectedProductCategory"
    InstanceName="CategoriesContextHandler.dataProvider"
    DataControl="CategoriesContextHandler"
    RequiresUpdateModel="true" Action="invokeMethod"
    MethodName="setSelectedProductCategory"
    IsViewObjectMethod="false"
    ReturnName="CategoriesContextHandler.methodResults.
    CategoriesContextHandler_dataProvider_setSelectedProductCategory_result">
    <NamedData NDName="categoryName" NDTType="java.lang.String"/>
```

```
<NamedData NDName="categoryId" NDType="oracle.jbo.domain.Number"/>
<events xmlns="http://xmlns.oracle.com/adfm/contextualEvent">
    <event name="selectCategory"/>
</events>
```

When the action binding is invoked, the event is created and the return of the method is added to the payload of the event.

When you configure an event map, JDeveloper creates an event map entry in the corresponding page definition file. [Example 26-11](#) shows the event map on the `homePageDef` page definition file that maps the `selectCategory` event from the `categoriestaskflow1` region to both the `MostPopularProductsByCategoriesExecuteWithParams` and the `ProductsByCategoriesExecuteWithParams` method bindings on the `products_productsTablePageDef` page definition file. Both of these consumers invoke methods that determine the products to display based on the parameters that are passed into it. The mapping is in the `homePageDef` page definition, as that is the parent container for both the `categoriestaskflow1` and the `MostPopularProductsFlow` regions.

Example 26-11 Event Map in Parent Page Definition File

```
<eventMap>
    <event name="selectCategory">
        <producer region="categoriestaskflow1">
            <consumer handler="products_productsTablePageDef.
                MostPopularProductsByCategoriesExecuteWithParams"
                region="MostPopularProductsFlow">
                <parameters>
                    <parameter name="category" value="${PayLoad['CategoryId']}"/>
                </parameters>
            </consumer>
            <consumer handler="products_productsTablePageDef.
                ProductsByCategoriesExecuteWithParams"
                region="MostPopularProductsFlow">
                <parameters>
                    <parameter name="category" value="${PayLoad['CategoryId']}"/>
                </parameters>
            </consumer>
        </producer>
    </event>
</eventMap>
```

26.5.4 What Happens at Runtime: Contextual Events

When both the event producer and the consumer are defined in the same page definition file, as the corresponding page is invoked and the binding container is created, the event is raised when the corresponding method or action binding is executed, when a value binding is set successfully, or when a range binding currency is set successfully.

For a method binding, the result of the method execution forms the payload of the event, and the payload is queued. The method binding passes the event to the binding container for dispatch. The event dispatcher associated with the binding container checks the event map (also in the binding container, as it is part of the same page definition file) for a consumer interested in that event and delivers the event to the consumer at the end of the lifecycle. The payload is then removed from the queue.

When the producer and consumer are in different regions, the event is first dispatched to any consumer in the same container, and then the event propagation is delegated to the parent binding container. This process continues until the parent or the topmost binding container is reached. After the topmost binding container is reached, the event is again dispatched to child binding containers that have regions with pages having producer set to wildcard "*" .

26.6 Adding ADF Model Layer Validation

In the model layer, ADF Model validation rules can be set for a binding's attribute on a particular page. When a user edits or enters data in a field and submits the form, the bound data is validated against any set rules and conditions. If validation fails, the application displays an error message.

Note that you don't need to add additional ADF Model validation if you have already set validation rules in the business domain layer of your entity objects. In a Business Components-based Fusion web application, unless you use data controls other than your application module data controls, you won't need to use ADF Model validation.

26.6.1 How to Add Validation

You set ADF Model validation on the page definition file. You define the validation rule, and set an error message to display when the rule is broken.

Table 26–1 describes the ADF Model validation rules that you can configure for a binding's attributes.

Table 26–1 ADF Model Validation Rules

Validator Rule Name	Description
Compare	Compares the attribute's value with a literal value
List	Validates whether or not the value is in or is not in a list of values
Range	Validates whether or not the value is within a range of values
Length	Validates the value's character or byte size against a size and operand (such as greater than or equal to)
Regular Expression	Validates the data using Java regular expression syntax
Required	Validates whether or not a value exists for the attribute

To create an ADF Model validation rule:

1. Open the page definition that contains the binding for which you want to create a rule.
2. In the Structure window, select the attribute, list, or table binding.
3. In the Property Inspector, select **More** and then **Edit Validation Rule**.
4. In the Edit Validation Rules dialog, expand the binding node, select the attribute name, and click **New**.
5. In the Add Validation Rule dialog, select a validation rule and configure the rule accordingly.
6. Select the **Failure Handling** tab and configure the message to display when the rule is broken.

26.6.2 What Happens at Runtime: Model Validation Rules

When a user submits data, as long as a submitted value is a non-null value or a string value of at least one character, then all validators on a component are called one at a time. Because the `f:validator` tag on the component is bound to the `validator` property on the binding, any validation routines set on the model are accessed and executed.

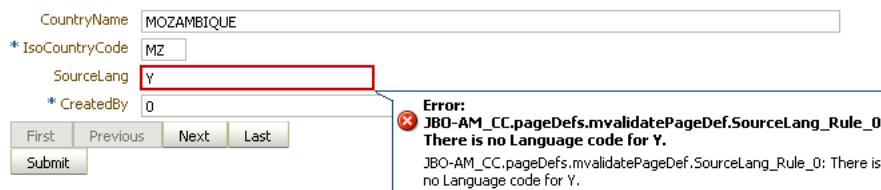
The process then continues to the next component. If all validations are successful, the Update Model Values phase starts and a local value is used to update the model. If any validation fails, the current page is redisplayed along with the error message.

26.7 Displaying Error Messages

When you use the Data Controls panel to create input components, JDeveloper inserts the `af:messages` tag at the top of the page. This tag can display all error messages in the queue for any validation that occurs on the server side, in a box offset by color. If you choose to turn off client-side validation for ADF Faces, those error messages are displayed along with any ADF Model error messages. ADF Model messages are shown first. Messages are shown within the `af:messages` tag, and with the associated components.

[Figure 26–4](#) shows the error message for an ADF Model validation rule, which states that the value the user entered is not acceptable.

Figure 26–4 Displaying Model Error Messages



You can display server-side error messages in a box at the top of a page using the `af:messages` tag. When you drop any item from the Data Controls panel onto a page as an input component, JDeveloper automatically adds this tag for you.

To display error messages in an error box:

1. In the Structure window, select the `af:messages` tag.

This tag is created automatically whenever you drop an input widget from the Data Controls panel. However, if you need to insert the tag manually, simply add the code, as shown in [Example 26–12](#), within the `af:document` tag.

Example 26–12 Messages Tag in a Page

```
<af:document>
  <af:messages globalOnly="false" />
  ...
</af:document>
```

2. In the Property Inspector set the following attributes:

- `globalOnly`: By default, ADF Faces displays global messages (that is, messages that are not associated with components), followed by individual component messages. If you wish to display only global messages in the box,

- set this attribute to `true`. Component messages will continue to display with the associated component.
- `Inline`: Specify whether to render the message list inline with the page or in a popup window.
 - `message`: The main message text that displays just below the message box title, above the list of individual messages.
3. Ensure that client-side validation has been disabled. If you do not disable client-side validation, the alert dialog will display whenever there are any ADF Faces validation errors and prevent propagation of the error to the server.

To disable client-side validation, add an entry for

`<client-validation-disabled>` and set it to `true` in the `trinidad-config.xml` file, as shown in [Example 26–13](#).

Example 26–13 Disabling Client-Side Validation in the Trinidad-config.xml

```
<?xml version="1.0" encoding="windows-1252"?>
<trinidad-config xmlns="http://myfaces.apache.org/trinidad/config">
  <skin-family>blafplus-rich</skin-family>
  <client-validation-disabled>true</client-validation-disabled>
</trinidad-config>
```

26.8 Customizing Error Handling

You can report errors using a custom error handler that extends the default `DCErrorHandlerImpl` class. You are not required to write any code to register your custom exception handler class. Instead, you select the root node of the `DataBindings.cpx` file in the Structure window, and then use the Property Inspector to set the `ErrorHandlerClass` property to the fully qualified name of the error handler you want it to use.

Your custom error handler can contain the following overridable methods:

- `reportException()`: Called to report any exception that occurs. It can be overridden to analyze reported exceptions.
- `getDisplayMessage()`: Returns the message that will be reported to JSF for each error that occurs. Returning `null` is the way your custom error handler signals that a given exception should not be reported to the client.
- `processMessage()`: Called every time an exception is transformed into a Faces message. It provides the ability to change the content of the message that will be displayed.

[Example 26–14](#) illustrates a custom error handler that extends the `DCErrorHandlerImpl` class.

Example 26–14 Custom Error Handler

```
package view.controller.fwkext;
import java.util.ArrayList;
import java.util.List;
import oracle.adf.model.binding.DCBindingContainer;
import oracle.adf.model.binding.DCErrorHandlerImpl;
import oracle.jbo.CSMMessageBundle;
import oracle.jbo.DMLConstraintException;
import oracle.jbo.JboException;
```

```
public class CustomErrorHandler extends DCErrorHandlerImpl {  
    List<ExceptionMapper> exceptionMapperList = new  
    ArrayList<ExceptionMapper>();  
    public CustomErrorHandler() {  
        this(true);  
    }  
    public CustomErrorHandler(boolean setToThrow) {  
        super(setToThrow);  
        exceptionMapperList.add(new DisableJboExceptionCodesMapper());  
    }  
    public void reportException(DCBindingContainer bc, Exception ex) {  
        for (ExceptionMapper mapper : exceptionMapperList) {  
            if (mapper.canMapException(ex)) {  
                ex = mapper.mapException(ex);  
            }  
        }  
        super.reportException(bc, ex);  
    }  
}
```

You must change the constructor to `MyErrorHandler()`. The exception error handler must have a default constructor, as shown in [Example 26–15](#).

Example 26–15 Default Constructor

```
ErrorHandlerClass="viewcontroller.MyErrorHandler"  
public MyErrorHandler()  
{  
    super(true);  
}
```

26.8.1 Writing an Error Handler to Deal with Multiple Threads

Oracle ADF constructs an instance of the custom error handler for each `BindingContext` object that is created. Because Oracle ADF serializes simultaneous web requests from the same logical end-user session, multiple threads generally will not use the same error handler at the same time. However, to guarantee a thread-safe custom error handler, use the `setProperty()` API on `JboException`. This method stores in the exception objects themselves any hints you might need later during the phase when exceptions are translated to JSF `FacesMessage` objects for display.

Designing a Page Using Placeholder Data Controls

This chapter describes how to create and use placeholder data controls. It shows you how to create placeholder data types, including master-detail relationships. It also describes how to create and import sample data.

This chapter includes the following sections:

- [Section 27.1, "Introduction to Placeholder Data Controls"](#)
- [Section 27.2, "Creating Placeholder Data Controls"](#)
- [Section 27.3, "Creating Placeholder Data Types"](#)
- [Section 27.4, "Using Placeholder Data Controls"](#)

27.1 Introduction to Placeholder Data Controls

Application development is typically divided into two separate processes: technical implementation and user interface design. More often than not, they are undertaken by separate teams with very different skill sets. The two teams can work together either in a *data-first* approach or a *UI-first* approach, or with some overlap between the two. With either approach, the teams usually work together iteratively, refining the application with each cycle.

In a data-first approach, the model, or data control is built first. Then the designer creates the layout and page flow by dragging and dropping the data controls onto pages as UI components. The model data is automatically bound to the components. This approach requires the data model to be available before the designer can proceed.

In a UI-first approach, the designer creates the layout using components from the Component Palette. When the data controls do become available, UI components are then bound to them. With this approach, you should be able to see most of the layout and page flows to make a development evaluation. However, until the data controls are available and bound to components, the application may not fully convey the intent of its design. For instance, an application that has a master-detail relationship is best reviewed when there is actual data that dynamically drives that relationship.

Placeholder data controls are easy-to-create, yet fully functional, stand-in data controls that can efficiently speed up the design-development process. UI designers can use placeholder data controls to create page layouts and page flows without the need to have real data controls available. These placeholder controls can be loaded with sample data to realistically simulate application execution for design evaluations. When the real data controls are ready, the UI components can be easily rebound to complete the application.

For many complex applications, the UI design may actually drive the development of the model, or data source. In this UI-first scenario, having placeholder data controls with sample data is essential to properly model the behavior of the application. In some cases, even if production data controls are available, UI designers may opt to use placeholder data controls because of their flexibility and ease of use.

Creating placeholder data controls is a purely declarative process and does not require coding. It does not require in-depth knowledge of the underlying model, data source technology, actual database schema, or any of the complex relationships in an actual production data control. Placeholder data controls do not require an existing data source or a database connection. You can define multiple data types with multiple attributes. You can also define nested master-detail hierarchies between data types. Placeholder data controls have the same built-in operations such as Execute, Next, and Create. An implicitly created named criteria item allows the user to create search forms as if view objects and view criteria were available.

Placeholder data controls can be used in many other situations. In addition for being used for design review and development, they can be used to develop realistic runtime mock-ups for usability studies, or for proof-of-concept requirements. They can be used to create demos when the data model is not yet ready.

27.2 Creating Placeholder Data Controls

You add placeholder data controls to a project using the New Gallery. After the placeholder data control has been created, it appears as a node in the Data Controls panel. It has a different icon than do standard data controls. Instead of an Operations node, the placeholder data control has a Built-in Operations node. Although the Built-in Operations node contains Commit and Rollback operations, these operations do not perform commits or rollbacks because there is not an actual data source for the data.

When a data control is initially created, it does not have any data types associated with it. You will need to manually create the data types as described in section [Section 27.3, "Creating Placeholder Data Types"](#)

27.2.1 How to Create a Placeholder Data Control

Placeholder data controls are defined at the project level in JDeveloper. You must already have created a project before you can create placeholder data controls.

To create a placeholder data control:

1. In the Application Navigator, right-click the project and choose **New**.
2. In the New Gallery, expand **Business Tier**, select **Data Controls** and then **Placeholder Data Control**, and click **OK**.
3. In the Placeholder Data Control dialog, as shown in [Figure 27-1](#), enter:
 - **Placeholder Name:** The name of the placeholder data control.
 - **Directory Name:** The package name that will be used to reference the placeholder data control.
 - **Description:** Optional description of the placeholder data control.

Figure 27–1 New Placeholder Data Control

4. Click OK.

27.2.2 What Happens When You Create a Placeholder Data Control

When you create a placeholder data control, the package you selected to contain the data control appears under the project node in the Application Navigator. A data control XML file *PlaceholderDataControl.xml* appears under the package, where *PlaceholderDataControl* is the name of the placeholder data control.

[Example 27–1](#) shows a sample file called *StoreFrontPlaceHolder.xml*, which was created when the *StoreFrontPlaceHolder* data control was created.

Example 27–1 Sample placeholderdatacontrol.xml

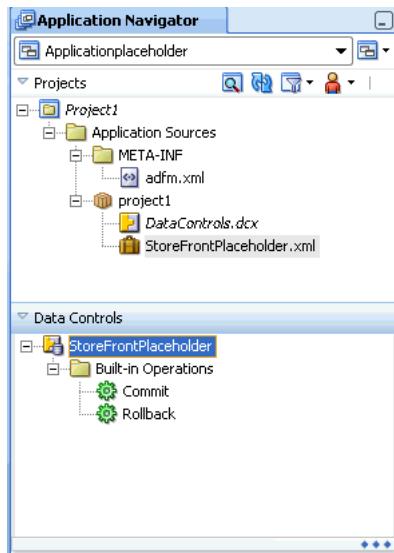
```
<?xml version='1.0' encoding='windows-1252' ?>
< AppModule
    xmlns="http://xmlns.oracle.com/placeholder"
    Name="StoreFrontPlaceholder" >
</ AppModule>
```

JDeveloper also creates a *DataControls.dcx* if it has not yet been defined, and adds entries for the placeholder data control, as shown in [Example 27–2](#).

Example 27–2 Placeholder Data Control entry in DataControls.dcx

```
<?xml version="1.0" encoding="UTF-8" ?>
< DataControlConfigs xmlns="http://xmlns.oracle.com/adfm/configuration"
    version="11.1.1.44.30" id="DataControls"
    Package="storefront">
    < PlaceholderDataControl SupportsTransactions="true" SupportsFindMode="true"
        SupportsResetState="true" SupportsRangeSize="true"
        SupportsSortCollection="true"
        FactoryClass="oracle.adf.model.placeholder.DataControlFactoryImpl"
        id="StoreFrontPlaceholder"
        xmlns="http://xmlns.oracle.com/adfm/datacontrol"
        Definition="storefront.StoreFrontPlaceholder"
        Package="storefront" />
</ DataControlConfigs>
```

In the Data Controls panel, the placeholder data control appears alongside other data controls in the root tree. A placeholder data control that does not yet have data types defined will have only the Commit and Rollback built-in operations available, as shown in [Figure 27–2](#).

Figure 27–2 Application Navigator and Data Controls Panel

27.3 Creating Placeholder Data Types

A standard data control obtains its data collections and attributes from its underlying data source in the model or business component layer. For example, an application module data control obtains its data collections from the view objects and associated database tables.

For a placeholder data control, instead of data collections, it has placeholder data types. A *placeholder data type* is analogous to a data collection. It can be dropped onto a page to create complex components such as forms, tables, and trees. It also has a set of attributes that can be dropped onto pages as individual components such as input text, output text, and select choice. Some attributes may be defined as LOVs.

When you first create a placeholder data control, it is devoid of any data types because there are no underlying data base tables for the placeholder data control to reference. You must declaratively create one or more placeholder data types. For each data type, you specify attribute names, types, default UI components, and other options. You can create multiple data types for a data control, similar to the multiple data collections in an application module.

After you have created a placeholder data type, it appears as a child node of the placeholder data control. It also has a Built-in operations node with the standard set of operations. It has a Named Criteria node that contains an `All Queriable Attributes` item that is analogous to the named view criteria of a view object in a standard data control. You can drag and drop the `All Queriable Attributes` item onto a page to create a query or quick query search form. In a standard data control, you can create multiple view criteria on a view object. Because there is no real view object in a placeholder data type, only one `All Queriable Attributes` item is available. For more information about query search forms, see [Chapter 25, "Creating ADF Databound Search Forms"](#).

You can create master-detail relationships between placeholder data types, similar to the master-detail data collections in a standard data control. You can drop master-detail data types onto pages to create master-detail forms and tables. For more

information on master-detail forms and tables, see [Chapter 22, "Displaying Master-Detail Data"](#).

JDeveloper allows you to reuse placeholder data types created for other placeholder data controls in the same project. When you are creating a data type, you can select an option to load existing data types from another placeholder data control. If you select the **Copy Data Type** option, the attributes from the imported data type will be added to the list of existing attributes.

You can also select an option to copy the sample data associated with the imported data type when the attributes are added.

27.3.1 How to Create a Placeholder Data Type

After you have created a placeholder data control, you can proceed to create data types. You define a name for the data type, then define each of its individual attributes. For each attribute, you then define its type, format, default UI component, and whether it should be an LOV.

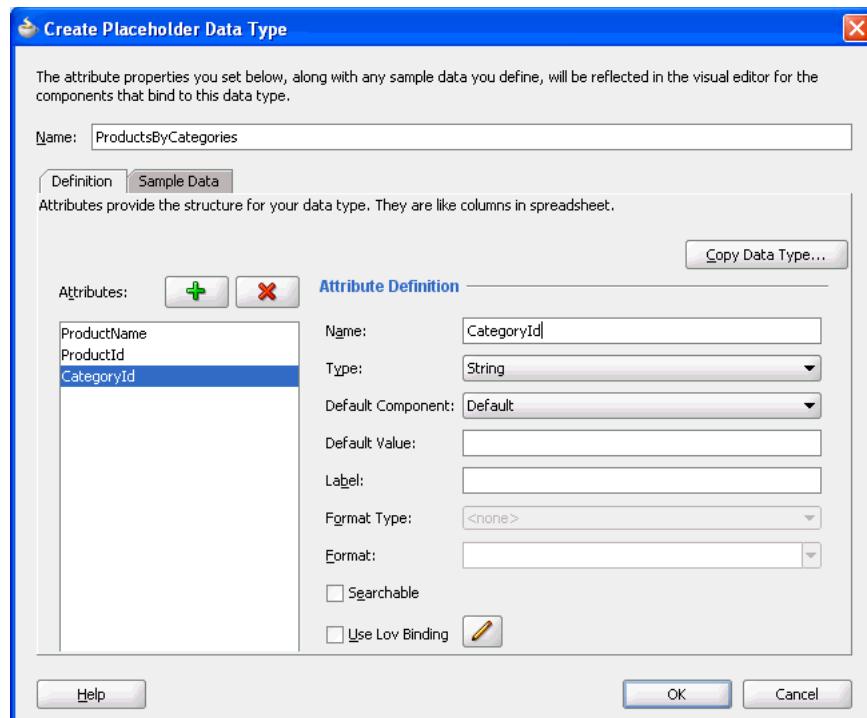
In order to simplify the process of creating placeholder data types, you can select from a list of four of the most common types: **String**, **Boolean**, **Date**, and **Number**. Because placeholder attributes are typed, you can create column labels and include UI control hints in the design.

If you have a sample data file in comma-separated-value or CSV format, you can automatically create all the attributes and load sample data using the sample data file import function. You do not need to create the attributes. JDeveloper will automatically create them for you from the format of the CSV file, which should be a comma-separated value list of column headings. The attributes default to type **String**. You can manually reset each attribute to another type as required. For instructions to import sample data, see [Section 27.3.6, "How to Add Sample Data"](#).

To create a placeholder data type manually:

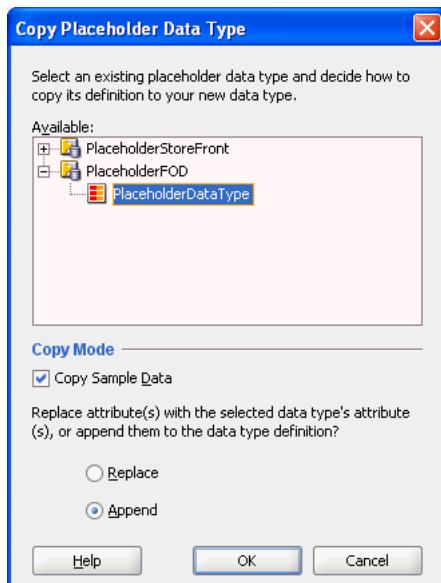
1. In the Data Controls panel, right-click the placeholder data control and choose **Create Placeholder Data Type**.

[Figure 27–3](#) shows the Create Placeholder Data Type dialog.

Figure 27–3 Create Placeholder Data Type Dialog

Note: If you had already added placeholder data types previously and want only to add or edit them, choose **Edit Placeholder Data Type** from the context menu instead. The dialog that appears will be the Edit Placeholder Data Type dialog. It has the same options as the Create Placeholder Data Type dialog.

2. If you already have data types defined for another placeholder data control, and you want to copy or append them, click **Copy Data Type**.
 - In the Copy Placeholder Data Type dialog, as shown in [Figure 27–4](#), select the placeholder data type you want. Select **Replace** to replace the current attributes with the attributes from the file, or select **Append** to add the attributes from the file to the list of current attributes.
 - Click the **Copy Sample Data** checkbox to load sample data from the file.
 - Click **OK**.

Figure 27–4 Copy Placeholder Data Type

3. In the Create Placeholder Data Type dialog, enter a name for the placeholder data type, and then in the **Attributes Definition** section, enter:
 - **Name:** Enter a name for the attribute.
 - **Type:** Select a type for the attribute from the dropdown list. The supported types are String, Boolean, Date, and Number.
 - **Default Component:** Select a default component for the attribute from the dropdown list. For an LOV, choose **Combo Box List of Values**
 - **Default Value:** Enter the initial value for the attribute.
 - **Label:** Enter a label for the attribute. The label will be used when the component is displayed.
 - **Format Type:** This field is enabled only when the type is Date or Number. Select a format type from the dropdown list.
 - **Format:** This field is enabled only when a format type has been selected. Select a format from the dropdown list.
 - **Searchable:** Select this checkbox to make the attribute searchable.
 - **Use Lov Binding:** Select this checkbox if you want the attribute to be an LOV. To configure the attribute, see [Section 27.3.3, "How to Configure a Placeholder Data Type Attribute to Be an LOV"](#).

Click the **Add** icon to add more attributes.
4. To add data, use the **Sample Data** tab. For that procedure, see [Section 27.3.6, "How to Add Sample Data"](#).
5. Click **OK**.

27.3.2 What Happens When You Create a Placeholder Data Type

When you create a placeholder data type, JDeveloper creates a *PlaceholderDataType.xml* file, where *PlaceholderDataType* is the name of the placeholder data type you had specified.

The *PlaceholderDataType.xml* file has the same format as a view object XML file. It includes the name of the view object and the name and values of each placeholder attribute that was defined.

Example 27–3 shows a *PlaceholderDataType.xml* for the Supplier data type. Two attributes were declarative defined: *Supplier_Id* and *Supplier_Name*.

Example 27–3 Sample Placeholder Data Type Suppliers.xml file

```
<?xml version='1.0' encoding='windows-1252' ?>
<ViewObject
    xmlns="http://xmlns.oracle.com/placeholder"
    Name="Suppliers"
    BindingStyle="OracleName"
    CustomQuery="true"
    ComponentClass="oracle.adf.model.placeholder.PlaceholderVOImpl"
    UseGlueCode="false" >
    <ViewAttribute
        Name="Supplier_Id"
        Type="oracle.jbo.domain.Number"
        PrimaryKey="true" >
    </ViewAttribute>
    <ViewAttribute
        Name="Supplier_Name"
        Type="java.lang.String" >
    </ViewAttribute>
    <StaticList
        Name="Suppliers"
        Rows="2"
        Columns="2" >
    </StaticList>
    <ResourceBundle >
        <PropertiesBundle
            xmlns="http://xmlns.oracle.com/bc4j"
            PropertiesFile="storefrontproject.StoreFrontProjectBundle" >
        </PropertiesBundle>
    </ResourceBundle>
</ViewObject>
```

Since a data type is similar to a data collection and is based on a view object, each data type will have a corresponding *PlaceholderDataType.xml* file.

JDeveloper also adds entries for each placeholder data type to the *PlaceholderDataControl.xml* file. For example, after the Suppliers data type has been created, the *StoreFrontPlaceholder.xml* file includes a new ViewUsage entry for the Suppliers data type, as shown in [Example 27–4](#).

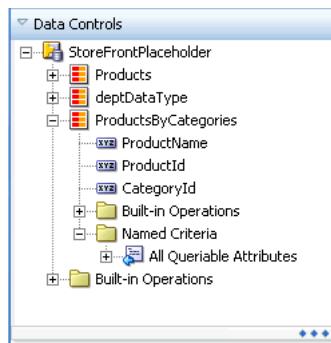
Example 27–4 Sample PlaceholderDataControl.xml File After Addition of Placeholder Data Type

```
<?xml version='1.0' encoding='windows-1252' ?>
< AppModule
    xmlns="http://xmlns.oracle.com/placeholder"
    Name="StoreFrontPlaceHolder" >
    <ViewUsage
        Name="Suppliers"
        ViewObjectName="storefrontproject.Suppliers" >
    </ViewUsage>
</ AppModule>
```

In the Data Controls panel, a placeholder data type node appears under the placeholder data control. Expanding the node reveals the presence of each of the attributes, the Built-in Operations node, and the Named Criteria node.

[Figure 27–5](#) shows a placeholder data control as it appears in the Data Controls panel.

Figure 27–5 Data Controls Panel Showing Placeholder Data Control



27.3.3 How to Configure a Placeholder Data Type Attribute to Be an LOV

A placeholder data type attribute can be configured to be a list of values (LOV). An LOV-formatted attribute binds to UI components that display dropdown lists or list picker dialogs. For more information about LOVs, see [Section 5.11, "Working with List of Values \(LOV\) in View Object Attributes"](#).

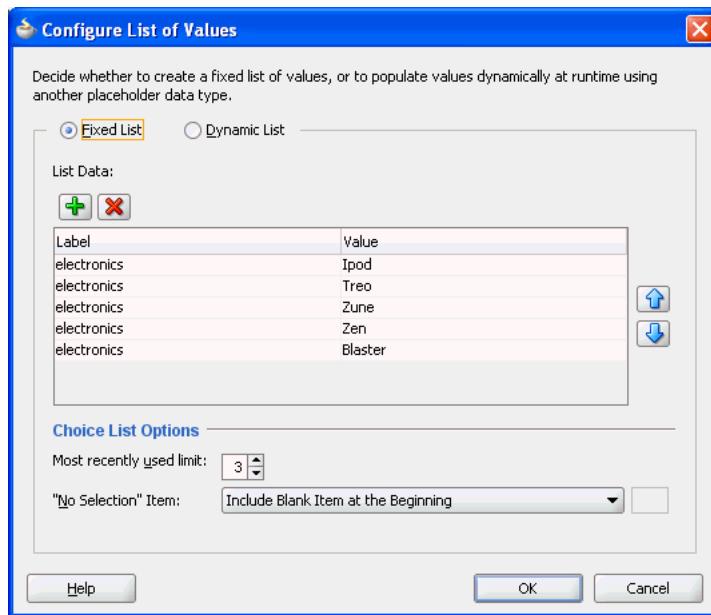
When you are creating a placeholder data type, you can select an option to bring up a dialog to configure that attribute to be an LOV.

If you have only one data source, you can only create a fixed list LOV. To create a dynamic list LOV, there must be more than one placeholder data type available to be the source.

To configure an attribute to be a fixed list LOV:

1. In the Data Controls panel, right-click the placeholder data control and choose **Create Placeholder Data Type** or **Edit Placeholder Data Type** from the context menu.
2. In the Create Placeholder Data type or Edit Placeholder Datatype dialog, click the **Use Lov Binding** checkbox.

The Configure List of Values dialog appears, as shown in [Figure 27–6](#).

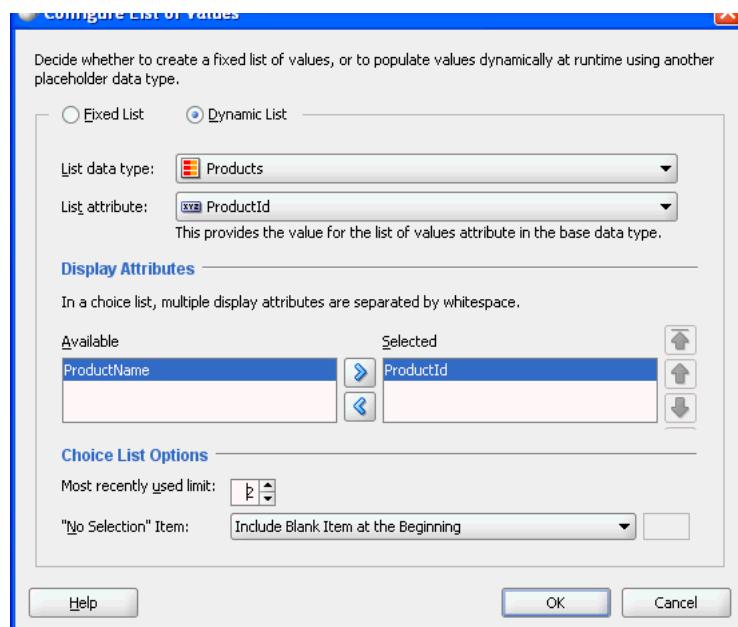
Figure 27–6 Configure List of Values Dialog for a Fixed List LOV

3. In the dialog, select **Fixed List**.
4. Click the **Add** icon to enable adding an entry to the list of values.
5. For each entry, enter a label and a value.
When the user selects an item from the list of values, the value entry will be entered into the input field.
6. Select the maximum number of the most recently used items that will be displayed in the dropdown list.
7. From the **No Selection Item** dropdown list, select an option for how you want the "no selection" item to be displayed.
For instance, selecting **Blank Item (First of List)** will display the "no selection" item as a blank at the beginning of the list.
8. Click **OK**.

To configure an attribute to be a dynamic list LOV:

1. In the Data Controls panel, right-click the placeholder data control and choose **Create Placeholder Data Type** or **Edit Placeholder Data Type**.
2. In the Create Placeholder Data type or Edit Placeholder Data type dialog, select the **Use Lov Binding** checkbox.

The Configure List of Values dialog appears, as shown in [Figure 27–7](#).

Figure 27-7 Configure List of Values Dialog for a Dynamic LOV

3. In the dialog, select **Dynamic List**.
4. Select the list data type with the source attribute. You must have a source placeholder data type for this selection to be available.
5. Select the list attribute.
6. Shuttle the attribute from the **Available** list to the **Selected** list.
7. Select the maximum number of the most recently used items that will be displayed in the dropdown list.
8. From the **No Selection Item** dropdown list, select an option for how you want the "no selection" item to be displayed. For instance, selecting **Blank Item (First of List)** will display the "no selection" item as a blank at the beginning of the list.
9. Click **OK**.

27.3.4 How to Create Master-Detail Data Types

You create master-detail relationships between data types in the same way you create master-detail hierarchies between tables. In a standard data control, you can use view links to define source and target view objects that would become the master and the detail objects. For more information about master-detail relationships, see [Chapter 22, "Displaying Master-Detail Data"](#).

Before you create a master-detail hierarchy, you must determine the data structure of the master data type and the data structure of the detail data type. You must also determine which attribute in the master will be the source for the detail data type.

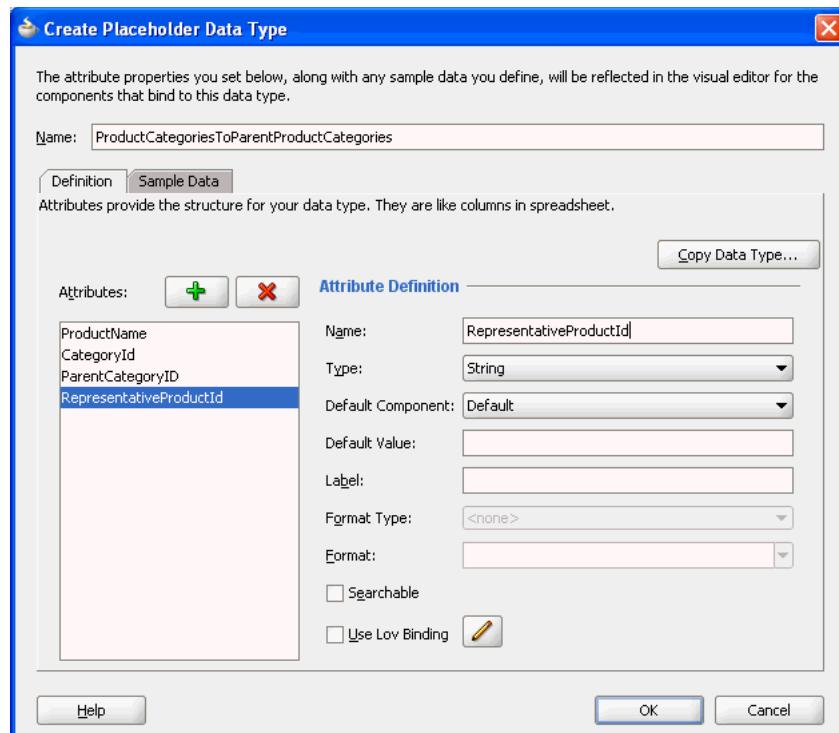
You first create a master data type and its attributes. Then you create a detail data type as a child of the master data type. You define the source attribute in the master data type that defines the relationship to the detail data type.

To create master-detail hierarchical data types:

1. Create a placeholder data type to be the master as described in [Section 27.3.1, "How to Create a Placeholder Data Type"](#), or select an existing data type to be the master. For example, the ProductsByCategories data type is the master.
2. In the Data Controls panel, right-click the master placeholder data type and choose **Create Placeholder Data Type**.

[Figure 27–8](#) shows the Create Placeholder Data Type dialog for entering detail data type attributes.

Figure 27–8 Create Placeholder Data Type Dialog

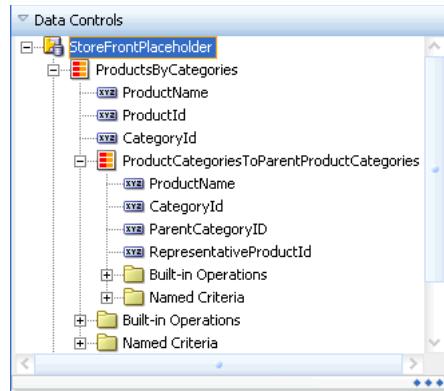


3. In the Create Placeholder Data Type dialog, enter a name for the detail data type.
4. The first attribute in the master data type appears in the **Attributes** section. This attribute provides the foreign key relationship.

Add attributes for the detail data type, copy data type attributes from another data type, or create attributes automatically by importing sample data from a CSV file. For the procedure to add attributes, see [Section 27.3.1, "How to Create a Placeholder Data Type"](#).

5. Click **OK**.

The Data Controls panel should display the detail data type as a child of the master data type, as shown in [Figure 27–9](#)

Figure 27–9 Master Detail Hierarchy in Placeholder Data Control

27.3.5 What Happens When You Create a Master-Detail Data Type

A master-detail relationship is implemented in the same way as is a standard master-detail relationship, using view object and view links. When you define placeholder data types in a master-detail hierarchy, JDeveloper creates a `DTLink.xml` file that contains metadata entries for view links that define that relationship. For more information about view links, see [Section 5.6, "Working with Multiple Tables in a Master-Detail Hierarchy"](#). For example, in the relationship between the master data type `Video` and the detail data type `Brand` associated with a key `dvdplayer`, JDeveloper creates a `DTLink.xml` file in the form of a view link file to define that relationship, as shown in [Example 27–5](#).

Example 27–5 DTLink.xml file for Master-Detail Data Type Relationships

```
<?xml version='1.0' encoding='windows-1252' ?>
<ViewLink
    xmlns="http://xmlns.oracle.com/placeholder"
    Name="DTLink" >
    <ViewLinkDefEnd
        Name="sourceEnd"
        Cardinality="1"
        Owner="project1.ProductsByCategories"
        Source="true" >
        <AttrArray Name="Attributes">
            <Item Value="project1.ProductsByCategories.ProductName" />
        </AttrArray>
    </ViewLinkDefEnd>
    <ViewLinkDefEnd
        Name="destEnd"
        Cardinality="-1"
        Owner="project1.ProductCategoriesToParentProductCategories" >
        <AttrArray Name="Attributes">
            <Item
Value="project1.ProductCategoriesToParentProductCategories.ProductName" />
        </AttrArray>
    </ViewLinkDefEnd>
</ViewLink>
```

27.3.6 How to Add Sample Data

If you intend to run an application using the placeholder data control, you will need to add sample data for execution. You can add sample data to the placeholder data type attributes manually or by importing the data from a CSV file.

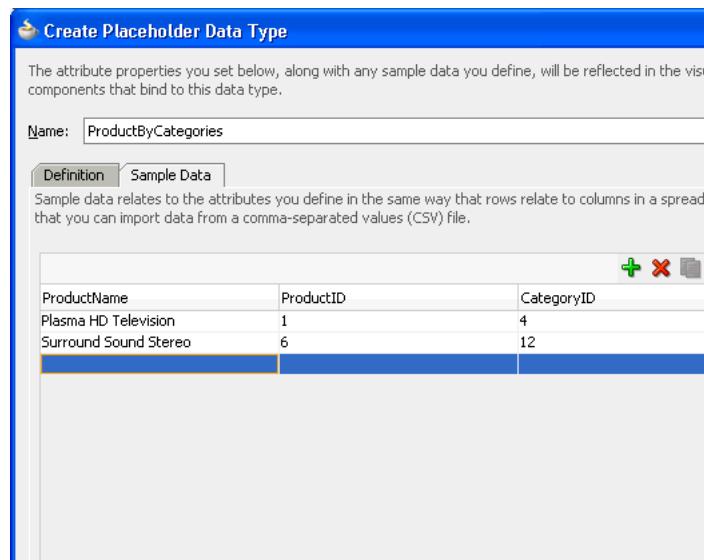
Before you begin to add sample data to a placeholder data type, you should have already created a placeholder data control and a placeholder data type. If you are entering the data manually, you should have the data ready. If you are loading the data from a CSV file, you need to have the location of the file.

To add sample data to placeholder data types manually:

1. In the Data Controls panel, right-click the placeholder data control and choose **Create Placeholder Data Type** or **Edit Placeholder Data Type** from the context menu.
2. In the Create Placeholder Data Type or Edit Placeholder Data Type dialog, click the **Sample Data** tab.
3. For each row of data, enter a value for each attribute that was defined. Click the **Add** icon to create each row.

For example, in [Figure 27–10](#), for the first row, Plasma HD Television was entered for the ProductName attribute, 1 was entered for the ProductID attribute, and 4 was entered for the CategoryID attribute.

Figure 27–10 Adding Sample Data



4. Click OK.

To import sample data from CSV files into placeholder data types:

1. In the Data Controls panel, right-click the placeholder data control and choose **Create Placeholder Data Type** or **Edit Placeholder Data Type** from the context menu.
2. In the Create Placeholder Datatype or Edit Placeholder Datatype dialog, click the **Sample Data** tab.

Tip: If you already have a CSV file for importing, you do not need to manually create the attributes for each column. JDeveloper automatically creates the attributes from the first row of the CSV file. For more information, see [Section 27.3.1, "How to Create a Placeholder Data Type"](#).

3. If you are also importing attributes, you must delete the default "attribute" in the first row.

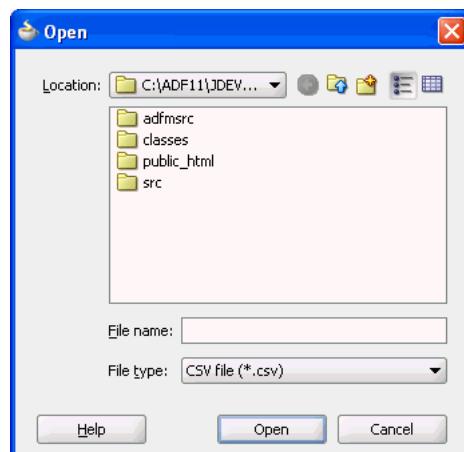
This default attribute appears when you first navigate to the **Sample Data** tab. If you do not remove this default attribute, JDeveloper will assume that this is a declaratively created attribute and will not import any other columns except for the first column.

4. Click **Import**.

In the Open dialog, navigate to and select the import file, and click **Open**, as shown in [Figure 27-11](#).

The data from the CSV file, including column heading and values, should appear as sample data.

Figure 27-11 Importing Placeholder Sample Data



5. Click **OK**.

27.3.7 What Happens When You Add Sample Data

Placeholder sample data, whether manually added using the dialog or from an imported CSV file, is stored in message bundle properties files within the placeholder data control packages. JDeveloper creates a text-based file for each data type that has sample data. The properties file name is `placeholderdatatypepnameMsgBundle.properties`. [Example 27-6](#) shows a sample data properties file for the Televisions data type that has three attributes of brand, size, and type.

Example 27-6 Sample Data Properties File `TelevisionMsgBundle.properties`

```
SL_0_0=sony
SL_0_1=42
SL_0_2=lcd
SL_1_0=panasonic
```

```
SL_1_1=50
SL_1_2=plasma
SL_2_0=mitsubishi
SL_2_1=60
SL_2_2=projection
```

27.4 Using Placeholder Data Controls

You use placeholder data controls in the same way you would use standard data controls. You can drag data types onto pages and use the context menus to drop the data types as forms, tables, trees, graphs, and other components. You can drop individual attributes onto pages as text, lists of values, single selections, and other components. You can use any of the built-in operations such as Create, Execute, and Next by dropping them as buttons, command links, and menu items.

You can work in several ways to take advantage of placeholder data controls:

- Build a page using the placeholder data controls and rebind to real data controls later.
- Build a page using components, bind them to placeholder data controls, and rebind to real data controls later.
- Build a page using some combination of components from the Component Palette, components from the placeholder data controls, and then binding or rebinding to the real data controls later.

27.4.1 Limitations of Placeholder Data Controls

You can use placeholder data controls in your application development in many situations. For most UI design evaluations, placeholder data controls should be able to fully substitute for real data controls.

There are a few limitations:

- Because data types are not tied to an underlying data source, the Commit and Rollback operations do not perform real transactions, nor do they update the cache.
- Placeholder data controls can only be created declaratively. You cannot create custom methods like you can with a real application module or data control.
- Placeholder data controls cannot be used when there is a need for custom data or filtering or custom code to fetch data. Placeholder data controls will disable those operations.

27.4.2 Creating Layout

Use the drag-and-drop feature to create the page using the placeholder data controls, any available real data controls, and components from the Component Palette. If you intend to run a page or application that requires real data, enter sample data for your placeholder data types. If you have a large amount of sample data, you may be able to create CSV files from the data source and load them into the data type. You may also use spreadsheets and other tools to create CSV sample data files.

27.4.3 Creating a Search Form

In a standard data control, you can create view criteria on view objects to modify the query. These view criteria are also used for drag-and-drop creation of query and quick query search forms. The named view criteria items appear under the Named Criteria node for a data collection. For more information about query and quick query search forms, see [Chapter 25, "Creating ADF Databound Search Forms"](#).

For placeholder data controls, there is also a Named Criteria node under each data type node. An automatically created All Queriable Attributes item appears under this node and can be used to drag and drop onto pages to create the query or quick query search forms.

27.4.4 Binding Components

Instead of building the page using the data controls, for instance, if you are unsure of the shape of your data, you can lay out the page first using the Component Palette and later bind it to the data types, attributes, or operations of the placeholder data controls.

27.4.5 Rebinding Components

After the final data controls are available, you can simply rebinding the components. You can select the component in the Structure window and use the context menu to open the relevant rebinding dialog. You can also drag and drop the data control item onto the UI component to initiate a rebinding editor. The rebinding procedures are the same whether the component was originally bound to a placeholder data control or a standard data control.

For more information about rebinding components, see [Chapter 20, "Creating a Basic Databound Page"](#) and [Chapter 21, "Creating ADF Databound Tables"](#)

27.4.6 Packaging Placeholder Data Controls to ADF Library JARs

A useful feature of placeholder data controls is that they allow parallel development and division of labor among developers and designers. You may be able to leverage that further by packaging placeholder data controls into reusable components as ADF Library JARs. ADF Libraries are JARs that have been packaged to contain all the necessary artifacts of an ADF component. For more information about reusable components and the ADF Library, see [Chapter 31, "Reusing Application Components"](#). You can create libraries of placeholder data controls and distribute them to multiple designers working on the same UI project. Because they are lightweight, you can even use them in place of available real data controls for the earlier phases of UI design.

Part V

Completing Your Application

Part V contains the following chapters:

- [Chapter 28, "Adding Security to a Fusion Web Application"](#)
- [Chapter 29, "Testing and Debugging ADF Components"](#)
- [Chapter 30, "Refactoring a Fusion Web Application"](#)
- [Chapter 31, "Reusing Application Components"](#)
- [Chapter 32, "Deploying Fusion Web Applications"](#)

28

Adding Security to a Fusion Web Application

This chapter describes how to use Oracle ADF Security in your Fusion web application to handle authentication and authorization when you test your application in JDeveloper and later when you deploy the application to Oracle Application Server.

This chapter includes the following sections:

- [Section 28.1, "Introduction to ADF Security for Fusion Web Applications"](#)
- [Section 28.2, "Choosing ADF Security Authentication and Authorization"](#)
- [Section 28.3, "Defining Users and Roles for a Fusion Web Application"](#)
- [Section 28.4, "Defining ADF Security Access Policies"](#)
- [Section 28.5, "Handling User Authentication in a Fusion Web Application"](#)
- [Section 28.6, "Performing Authorization Checks in a Fusion Web Application"](#)
- [Section 28.7, "Getting Other Information from the Oracle ADF Security Context"](#)
- [Section 28.8, "Testing ADF Security with Integrated WLS in JDeveloper"](#)
- [Section 28.9, "Preparing for Deployment to a Production Environment"](#)

28.1 Introduction to ADF Security for Fusion Web Applications

Oracle ADF implements a Java Authentication and Authorization Service (JAAS) security model through integration with the Oracle Platform Security for Java implementation of the JAAS service. The JAAS model implements authentication in a pluggable fashion, so applications can remain independent from the underlying authentication technology. However, JAAS still requires custom code at the application level that makes implementing authorization more difficult. Oracle ADF Security simplifies the implementation of a JAAS authorization model by exposing it in a declarative fashion on various Fusion web application resources that JDeveloper supports.

This declarative support for securing Fusion web application resources is not based on the URL mapping of a security constraint. Oracle ADF Security lets you define an *access policy* for a variety of application resources. For example, you can control access to a particular task flow based on the access right grants that you make in the *policy store* for the ADF task flow. During development, this policy store is file-based, with access right grants stored in the `jazn-data.xml` file, whereas the *identity store* can be file-based or LDAP-based, with grants stored in an LDAPv3-compliant directory, such as Oracle Internet Directory.

At runtime, the grants you define for Fusion web application resources will be enforced using standard JAAS permission authorization to determine the user's access

rights. If your application requires it, you can use Expression Language (EL) to surface server-side security enforcement directly on the user interface. You might use an EL expression to perform runtime permission checks within the web page to hide components that should not be visible to the user with insufficient privileges. You can also use Oracle ADF Security's implementation of expressions that are specific to the ADF task flow and ADF page definition to perform permission checks to prevent the user from being able to access a task flow or web page.

Within the Oracle ADF framework, JAAS-based security is enforced by the ADF binding servlet filter and the ADF Model layer of the application. The filter is configured to protect ADF resources and requires the current user to have sufficient access right grants to view the ADF resource.

The use of Oracle ADF Security enables web applications to easily adjust to real-world business security requirements, because rather than securing paths, you secure the actions of ADF resources with JAAS. JAAS-based Oracle ADF Security provides:

- More granular declarative security

Because Java EE security is URL-based or page-based, it is not possible to have a finer level of access control. With Oracle ADF Security, different roles can perform different levels of activity at the same URL.

- Simplified permission assignment by using hierarchical roles, which allow for the inheritance of permissions

While Java EE security roles that are used by Java EE security constraints are flat, JAAS permissions reference enterprise roles, which can be nested and may be mapped to application-specific roles.

- Utility methods for use in EL expressions to access ADF resources in the security context

You can use the Oracle ADF Security EL expression utility methods to determine whether the user is allowed to perform a known operation. For example, you can determine whether the user is allowed to access a particular task flow.

Note: When you want to understand the security features of Oracle WebLogic Server, see *Oracle WebLogic Server Understanding WebLogic Server*.

28.2 Choosing ADF Security Authentication and Authorization

JDeveloper and Oracle ADF Security allow you to choose to enable authentication and authorization separately. You may choose to:

- Enable the ADF authentication servlet and also enforce ADF authorization. Enforcing authorization means you intend to control access to Fusion web application resources by assigning access right grants to application roles.

Note: When enforcing ADF authorization, be aware that all ADF resources will be secure by default. Thus the step of configuring a policy store is necessary to make these resources accessible to authenticated users.

- Enable only the ADF authentication servlet. Enabling the ADF authentication servlet means that you want to support dynamic authentication, but that you intend to define container-managed security constraints to secure web pages.

Oracle ADF Security supports these two choices to leave you the option to rely on container-managed security constraints and still be able to support dynamic login using the ADF authentication servlet. In this case, the servlet will require the user to log in the first time a page in the application is accessed. The servlet will also redirect the user to the requested page when the user has sufficient access rights as defined by security constraints. In this case, the developer is responsible for defining security constraints for web pages.

When you prefer to define fine-grained security using ADF authorization, that option is also available. In this case, when choosing to enforce ADF authorization, you are responsible for specifying access right grants needed to make the ADF resources accessible to users. This step requires configuring a policy store that consists of grants made to specific application roles. You should plan on performing the following tasks:

1. Define application roles and assign members to those roles.

As a convenience, the Configure ADF Security wizard lets you define the `test-all` role to enable the members of this role to access all ADF resources. For details about the `test-all` role, see [Section 28.3.1, "How to Enable the test-all Application Role in JDeveloper"](#).

2. Make Permission grants to the roles in the policy editor for ADF resources.

When you enable the `test-all` role, the Configure ADF Security wizard will also make automatic "View" grants to the `test-all` role for the ADF resources that your application contains. You can then associate members with the `test-all` role, as described in [Section 28.3.3, "How to Configure the Identity Store with Test Users in JDeveloper"](#). As the application roles become known, you can eventually replace all these automatic grants with ones that define the production application's roles (rather than the `test-all` role).

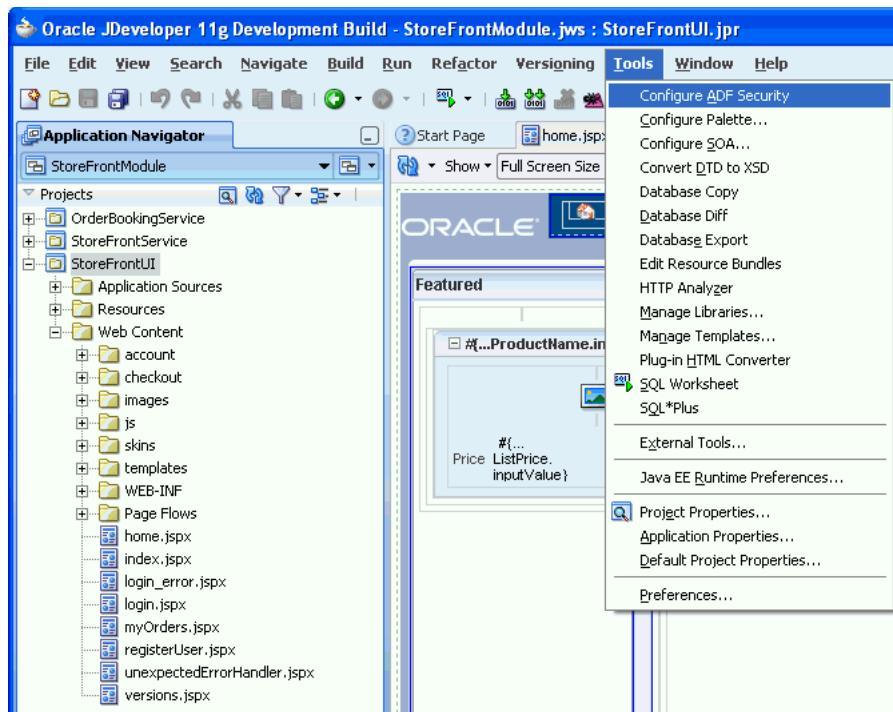
To manage these choices for authentication and authorization, you use the Configure ADF Security wizard in JDeveloper.

To launch the Configure ADF Security wizard:

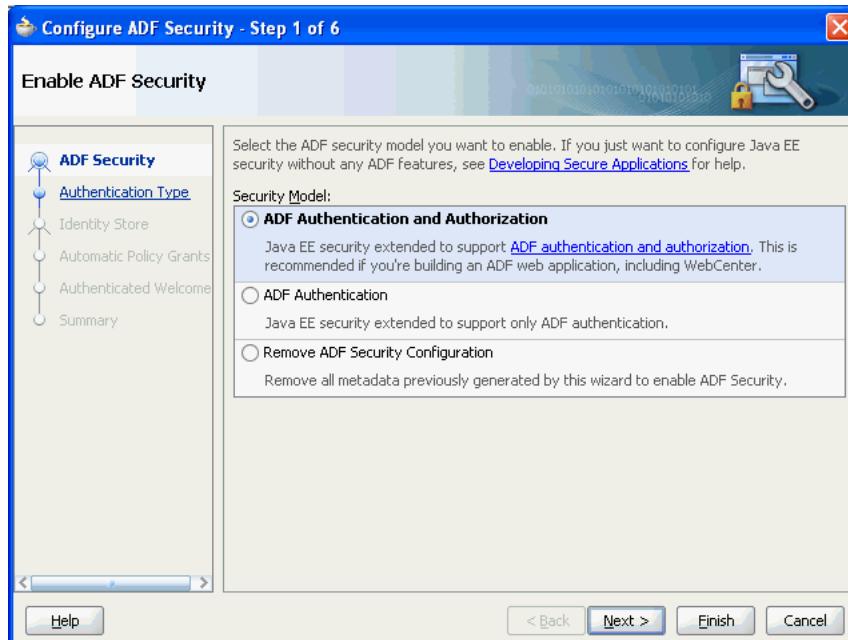
1. In the Application Navigator, select the user interface project that contains the web pages and `web.xml` file.

Caution: The Configure ADF Security wizard updates the `web.xml` file in the context of the project that appears selected in the Application Navigator. Before you run the wizard, make sure to select the user interface project in the Application Navigator. You can accomplish this by selecting any file or folder in the user interface project that you want to secure.

2. In the JDeveloper toolbar, choose **Configure ADF Security** from the **Tools** menu, as shown in [Figure 28–1](#).

Figure 28–1 Launching the Configure ADF Security Wizard from the Tools Menu

The first page of the wizard lets you choose one of three options, with the default option set to enable **ADF Authentication and Authorization**, as shown in Figure 28–2.

Figure 28–2 The Default Option in the Configure ADF Security Wizard

When you run the wizard with the default option selected, it will enable the ADF authentication servlet to handle user authentication dynamically. The wizard will add two constraints, `allPages` and `adfAuthentication`, to the `web.xml` file. The `allPages` constraint is mapped to the '/' URL. This mapping means that the first time

the user accesses any page, login will be triggered. For a complete description of how ADF Security handles authentication, see [Section 28.2.8, "What Happens at Runtime: How Oracle ADF Security Handles Authentication"](#).

Note: If you remove this `allPages` constraint from the `web.xml` file, you could provide a login link or button to explicitly trigger login. You could also have a link or button to perform logout.

The default option of the wizard will also enforce authorization. When you enforce authorization, all task flows and the web pages they contain and, additionally, any web page not contained by a task flow that is defined by an associated ADF page definition will be secure by default. This means that these web pages associated with ADF resources will require a security policy to grant access or they will remain inaccessible to the authenticated user. When you enforce authorization of ADF resources it means that you intend to secure access to ADF application resources and then allow certain application roles access to these resources by defining appropriate grants in the policy store. For a description of how ADF Security enforces authorization, see [Section 28.2.9, "What Happens at Runtime: How Oracle ADF Security Handles Authorization"](#).

As a convenience, the wizard prompts you to provide automatic grants to the `test-all` role. By using the wizard to grant to the `test-all` role, you can postpone defining explicit grants to ADF resources until you are ready to refine the access policy of your application. Then, as application development progresses and the content of the application becomes well-established, you can replace grants to the `test-all` role with the production application roles and explicit permission grants. The explicit grant establishes the necessary privilege (for example, `View on a page`) to allow users to access these resources. For details about ADF security policies, see [Section 28.4, "Defining ADF Security Access Policies"](#).

If you prefer instead not to enforce authorization of ADF resources, you can choose the **ADF Authentication** option. When you run the wizard with this option selected, it will enable the ADF authentication servlet to handle user authentication dynamically. Since the wizard does not enforce authorization for this option, no permission checking is performed for ADF resources. Therefore, once the user logs in, all ADF resources will be considered public and consequently all web pages will be accessible to the authenticated user.

The remaining option, **Remove ADF Security Configuration**, lets you disable the ADF authentication servlet for as long as you require. This option will appear enabled only if you have configured ADF Security with the wizard. You may select this option with the intention of reenabling ADF Security at any time. When you run the wizard with this option selected, the wizard removes ADF-specific metadata in the `web.xml` file and `adf-config.xml` file. The wizard specifically does not alter the policy store of the application, which contains the Permission metadata that application developers defined for ADF resources. For example, if the application uses the standard XML provider as the policy store, the `jazn-data.xml` file will remain unchanged by this option. This means that you can return to the wizard at any time, select the **ADF Authentication and Authorization** option, and reenable security against your application's existing policy and identity stores.

In summary, the Configure ADF Security wizard provides these three options, which you can select as required:

- **ADF Authentication and Authorization** enables the ADF authentication servlet and also enforces ADF authorization so that the policy store that your application

defines can be used for permission checking. This option assumes that you have defined application roles and assigned grants to those roles for specific ADF resources. As a convenience for making resources accessible after running the wizard, the wizard prompts you to provide automatic grants to the `test-all` role in the Configure ADF Security wizard.

- **ADF Authentication** enables the ADF authentication servlet to require the user to log in the first time a page in the application is accessed. Since the wizard disables ADF authorization, permission checking is not performed whether or not security policies exist for ADF resources. Once the user is logged in, all web pages containing ADF resources will be available to the user.
- **Remove ADF Security Configuration** disables the ADF authentication servlet and prevents ADF Security from checking policy grants without altering the existing policy store. In this case, you may require users to log in, become authenticated, and test access rights against URL security constraints using standard Java EE security. Note that running the wizard with this option disables fine-grained security against ADF resources.

Note: For more information about the choices you have for working with Java EE security, see "Roadmap for Developing Secure Applications" in the "Developing Secure Applications" section of the JDeveloper online help.

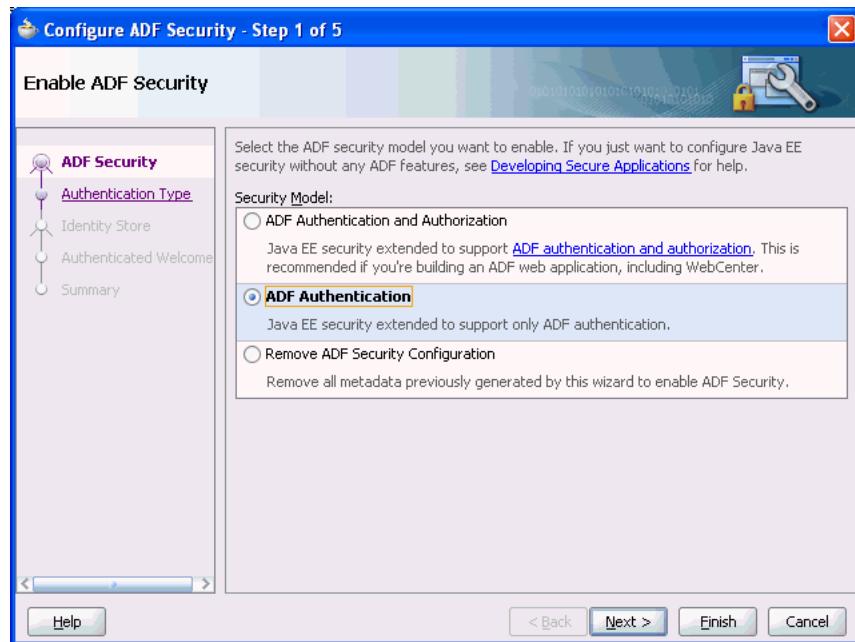
28.2.1 How to Enable Only ADF Authentication

After you run the Configure ADF Security wizard with the **ADF Authentication** option selected in the ADF Security page, you will have configured a security constraint against the ADF authentication servlet and enabled dynamic authentication. The wizard updates all security-related configuration files and ensures that the ADF authentication servlet enforces user authentication. With this option selected, Oracle ADF Security will *not* check Permission metadata that may have been created for ADF task flows and ADF bindings; and all ADF resources will be available to the user.

To enable authentication without resource authorization:

1. In the Application Navigator, select the user interface project that contains the pages that you want to secure. Do not select the data model project or any other project.
2. From the Tools menu, choose **Configure ADF Security**, as shown in [Figure 28-1](#).
3. In the ADF Security page, select **ADF Authentication**, as shown in [Figure 28-3](#). Click **Next**.

This selection configures the ADF authentication servlet. This will allow the servlet to set the security context to create an authenticated *Subject* (a container object that represents the user), as described in [Section 28.2.8, "What Happens at Runtime: How Oracle ADF Security Handles Authentication"](#).

Figure 28–3 Enabling Only Authentication in the Configure ADF Security Wizard

4. In the Authentication Type page, select the authentication type that you want your application to use when the user submits their login information. Click **Next**.

The most commonly used types of authentication are **HTTP Basic Authentication** and **Form-Based Authentication**. Basic authentication uses the browser login dialog for the user to enter a username and password. Note that with Basic authentication, the browser caches credentials from the user, thus preventing logout. *Basic authentication* is useful when you want to test the application without requiring a custom login page. *Form authentication* allows the application developer to specify a custom login UI.

If you select **Form-based Authentication**, you can also select **Generate Default Pages** to allow the wizard to generate a default login and error page. Alternatively, you can create your own pages.

For more information about handling user login, see [Section 28.5, "Handling User Authentication in a Fusion Web Application"](#).

5. In the Identity Store page, select the type of repository to store the user and role information for authentication purposes. Click **Next**.

The repository associated with the security provider can be an XML file or a directory service. Both repository types are supported by JDeveloper's Integrated WLS. However, when you choose the LDAP identity store, you must configure the LDAP store outside of JDeveloper. When you choose the application XML identity store, you can configure user and role information within JDeveloper using the editor for the `jazn-data.xml` file. This step will allow you to add users to the identity store so that you can run the application in JDeveloper and authenticate test users.

For more information about adding users to the identity store, see [Section 28.3, "Defining Users and Roles for a Fusion Web Application"](#).

6. In the Authenticated Welcome page, optionally specify a web page that the application will use to redirect the authenticated Subject to. Click **Next**.

Use to direct the user to a specific page after they log in. The application will use the specified web page only when a destination page is not specified as a parameter to the ADF authentication servlet. Typically, the application will specify a destination page as part of the `success_url` parameter. In particular, one is specified by Oracle ADF when the application attempts to access a web page that requires ADF authorization and the user is not yet logged in, as described in [Section 28.2.8, "What Happens at Runtime: How Oracle ADF Security Handles Authentication"](#). If you do not specify a redirect web page, the servlet will direct the user back to the page from which login was initiated.

7. In the Summary page, review your selections and click **Finish**.

28.2.2 What Happens When You Choose Not to Enforce Authorization

[Table 28–1](#) shows which files the Configure ADF Security wizard updates when you complete the wizard with the **ADF Authentication** option selected.

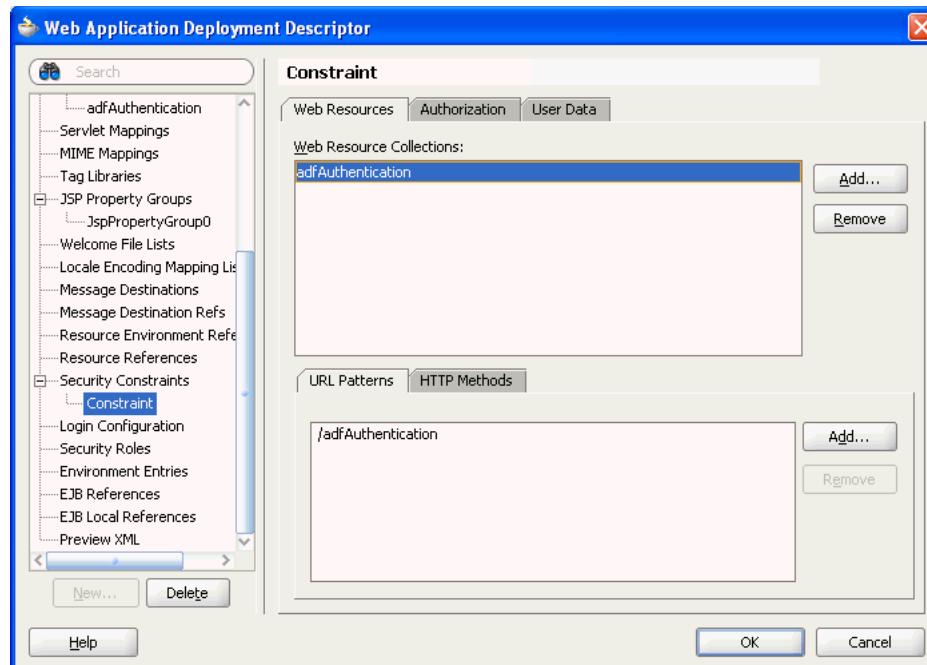
Table 28–1 Files Updated for ADF Authentication

File	File Location	Configuration Performed by the Configure ADF Security Wizard
<code>web.xml</code>	/public_html/WEB-INF folder relative to the user interface project	<ul style="list-style-type: none"> ▪ Oracle JpsFilter filter definition required for ADF authorization. ▪ Oracle adfAuthentication servlet definition. ▪ Servlet mapping for the ADF authentication servlet. ▪ Security constraint on the ADF authentication servlet. ▪ Login configuration. ▪ Required security roles, including the role <code>valid-users</code>, which is used to trigger authentication.
<code>adf-config.xml</code>	/.adf/META-INF folder relative to the web application workspace	<ul style="list-style-type: none"> ▪ JAAS security context. ▪ The use of the ADF authentication servlet is enforced (<code>authenticationRequire</code> property in the <code><JaasSecurityContext></code> element is set to <code>true</code>). This property is used in J2SE applications to trigger displaying a login dialog. ▪ The use of security policies for ADF resources are ignored and no permission checking is performed (the <code>authorizationEnforce</code> parameter in the <code><JaasSecurityContext></code> element is set to <code>false</code>).

Table 28–1 (Cont.) Files Updated for ADF Authentication

File	File Location	Configuration Performed by the Configure ADF Security Wizard
jps-config.xml	%domain.home%/config folder relative to the domain.home environment variable	<ul style="list-style-type: none"> ▪ Oracle Platform Security context for the credential store. ▪ Oracle Platform Security context for the policy store. ▪ Oracle Platform Security context for anonymous user, which contains the anonymous service instance and the anonymous login module.
weblogic.xml	/src/META-INF folder relative to the web application workspace	<ul style="list-style-type: none"> ▪ Role mapping of the valid-users security role to the Oracle Platform Security principal, users.
jazn-data.xml	./src/META-INF folder relative to the web application workspace	<ul style="list-style-type: none"> ▪ Default jazn.com realm name for the XML file-based identity store.

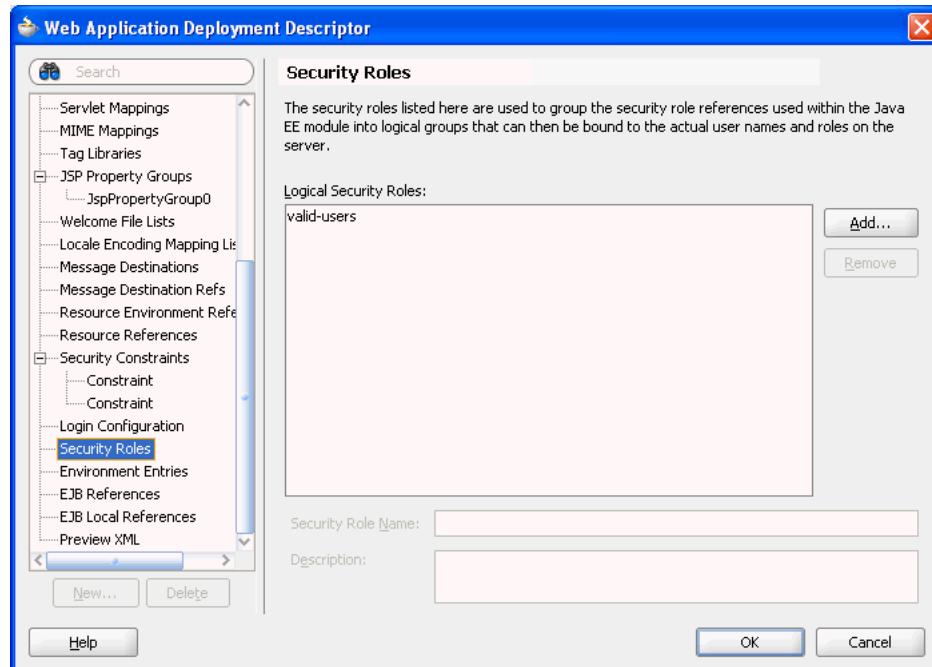
As authentication is delegated to the web container, the wizard updates the web.xml file to define an adfAuthentication resource that uses a URL pattern against the ADF authentication servlet, as shown in [Figure 28–4](#). This constraint allows for the definition of a single standard URL that can be used as a login or logout link throughout the application. Java EE container-managed security defines a standard method for login. There is no standard to log out of an application, so while the login process is delegated to the container, the logout process is handled by the ADF authentication servlet itself.

Figure 28–4 Security Constraint Defined by Configure ADF Security Wizard

Note: You must not delete the adfAuthentication constraint from the web.xml file. Deleting it would prevent users from logging in to the application in the manner supported by Oracle ADF. Any other static security constraints would continue to work and invoke a login if required.

Because every user of the application is required to be able to log in, the security constraint defined against the ADF authentication servlet should allow all users to access this web resource. As such, the security role associated with the constraint should encompass all users. To simplify this task, the Java EE valid-users role is provided, as shown in [Figure 28–5](#). The the web.xml file maps this role to a special authenticated-role principal defined by Oracle Platform Security. This mapping ensures that every user will have this role because every properly authenticated user will have the authenticated-role principal added to their Subject by Oracle Platform Security, as described [Section 28.2.3, "What You May Need to Know About the valid-users Role"](#). From a security perspective, the valid-users role lets you control access to web resources using only security constraints. The end result of this mapping relies entirely on Java EE security and does not involve JAAS Permissions.

Figure 28–5 valid-users Role Defined by Configure ADF Security Wizard



You may also use this page to add further security constraints on web resources used by the application, as described in "Securing Web Pages Using a Security Constraint" in the "Developing Secure Applications" section of the JDeveloper online help.

28.2.3 What You May Need to Know About the valid-users Role

The valid-users role is a Java EE security role defined by ADF Security to trigger the authentication of the user by the security constraint on the adfAuthentication web resource defined in the web.xml file. The Configure ADF Security wizard updates the weblogic-application.xml file to map this ADF security role to the authenticated-role principal. The authenticated-role is a special principal

defined by Oracle Platform Security and serves a purpose similar to an enterprise role. At runtime, the principal is added automatically to a successfully authenticated Subject by Oracle Platform Security.

28.2.4 How to Enable ADF Authentication and Authorization

After you run the Configure ADF Security wizard with the **ADF Authentication and Authorization** option selected in the ADF Security page, you will have configured a security constraint against the ADF authentication servlet, enabled dynamic authentication, and enabled ADF permission checking. The wizard updates all security-related configuration files and ensures that ADF resources are secure by default.

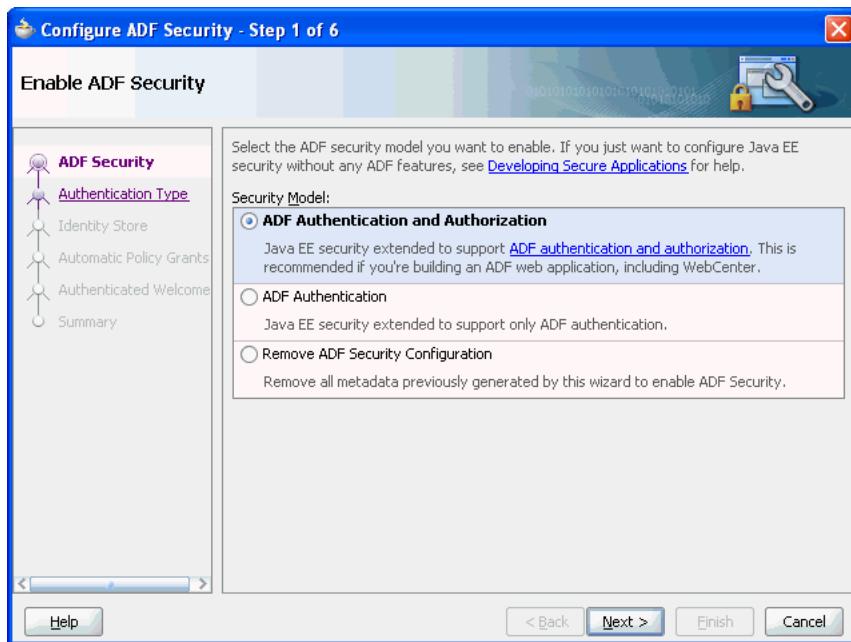
When you run the Configure ADF Security wizard to enable authorization of ADF resources, you are not required to have a policy store in place. Permission checking can still take place using the test-all role that you can enable in the wizard. If you define this role, and associate those members with the role, members from the identity store will be granted automatic view access on ADF resources, as described in [Section 28.3, "Defining Users and Roles for a Fusion Web Application"](#). This allows you to run and test your application before creating the policy store.

Eventually, you will want to make grants to customize the ADF resource access rights for members of actual application roles. For complete information about how to customize the policy store so that your application has fine-grained control over access privileges on ADF resources, see [Section 28.4, "Defining ADF Security Access Policies"](#).

To enable authentication and authorization of ADF resources:

1. In the Application Navigator, select the user interface project that contains the pages you want to secure. Do not select the data model project or any other project.
2. From the Tools menu, choose **Configure ADF Security**.
3. In the ADF Security page, select **ADF Authentication and Authorization**, as shown in [Figure 28–6](#).

This selection configures the ADF authentication servlet. This configuration will allow the servlet to set the security context and check for ADF authorization privileges.

Figure 28–6 Enabling Authorization in the Configure ADF Security Wizard

4. In the Authentication Type page, select the authentication type that you want your application to use when the user submits their login information. Click **Next**.

The most commonly used types of authentication are **HTTP Basic Authentication** and **Form-Based Authentication**. Basic authentication uses the browser login dialog for the user to enter a username and password. Note that with Basic authentication, the browser caches credentials from the user, thus preventing logout. Basic authentication is useful when you want to test the application without requiring a custom login page. Form authentication allows the application developer to specify a custom login UI.

If you select **Form-based Authentication**, you can also select **Generate Default Pages** to allow the wizard to generate a default login and error page. Alternatively, you can create your own pages.

For more information about these user login options, see [Section 28.5, "Handling User Authentication in a Fusion Web Application"](#).

5. In the Identity Store page, select the type of repository to store the user and role information for authentication purposes. Click **Next**.

The repository associated with the security provider can be an XML file or a directory service. Both repository types are supported by JDeveloper's Integrated WLS. However, when you choose the LDAP identity store, you must configure the LDAP store outside of JDeveloper. When you choose the application XML identity store, you can configure user and role information within JDeveloper using the editor for the `jazn-data.xml` file. This step will allow you to add users to the identity store so that you can run the application in JDeveloper and authenticate test users.

For more information about adding users to the identity store, see [Section 28.3, "Defining Users and Roles for a Fusion Web Application"](#).

6. In the Automatic Policy Grants page, select whether to make ADF resources public without requiring application developers to first define ADF policy grants. Click **Next**.

When you enable automatic policy grants, a "View" grant will be made to all members of the `test-all` application role. This option provides a convenient way to run and test application resources without the restricted access that ADF authorization enforces. To enable this option, you can select either **Grant to Existing Objects Only** or you can select **Grant to All Objects**. The choice between existing objects and all objects lets you test your application resources either as developers add them (*all* objects) or you can limit testing so that new resources remain inaccessible (only *existing* objects). After you enable the automatic policy grant in the wizard, you must add a test user to the `test-all` role. Alternatively, if you want to run the application without restriction, you can assign `anonymous-role` as a member of the `test-all` role. For details about adding members to the `test-all` role, see [Section 28.3.3, "How to Configure the Identity Store with Test Users in JDeveloper"](#).

Otherwise, when you are ready to configure ADF policy grants to secure ADF resources, select **No Automatic Grants**.

Note: If you select **No Automatic Grants**, all ADF resources will be secure by default. Thus the step of configuring a policy store is necessary to make the web pages and task flows accessible to authenticated users. For details about the granting access to ADF resources, see [Section 28.4, "Defining ADF Security Access Policies"](#).

7. In the Authenticated Welcome page, optionally, select **Redirect Upon Successful Authentication** when you want to direct the user to a specific web page after they log in. Click **Next**.

If not selected, the ADF authentication servlet directs the user back to the page from which the login was initiated.

Note that if the web page you specify contains ADF Faces components, you must define the page in the context of `/faces/`. For example, the path for `adffaces_welcome.jsp` would appear in the **Welcome Page** field as `/faces/adffaces_welcome.jsp`.

8. In the Summary page, review your selections and click **Finish**.

28.2.5 What Happens When You Choose to Enforce Authorization

[Table 28–2](#) shows which files the Configure ADF Security wizard updates.

Table 28–2 Files Updated for ADF Authentication and Authorization

File	File Location	Configuration Performed by the Configure ADF Security Wizard
web.xml	/public_html/WEB-INF folder relative to the user interface project	<ul style="list-style-type: none"> ▪ Oracle JpsFilter filter definition required for ADF authorization. ▪ Oracle adfAuthentication servlet definition. ▪ Servlet mapping for the ADF authentication servlet. ▪ Security constraint on the ADF authentication servlet. ▪ Login configuration. ▪ Required security roles, including the role, valid-users, which is used to trigger authentication.
adf-config.xml	/.adf/META-INF folder relative to the web application workspace	<ul style="list-style-type: none"> ▪ JAAS security context. ▪ The use of the ADF authentication servlet is enforced (authenticationRequire parameter in the <JaasSecurityContext> element is set to true). This property is used in J2SE applications to trigger displaying a login dialog. ▪ The use of ADF Security metadata is enforced for permission checking (the authorizationEnforce parameter in the <JaasSecurityContext> element is set to true).
jps-config.xml	%domain.home%/config folder relative to the domain.home environment variable	<ul style="list-style-type: none"> ▪ Oracle Platform Security context for the credential store. ▪ Oracle Platform Security context for the policy store. ▪ Oracle Platform Security context for anonymous user, which contains the anonymous service instance and the anonymous login module.
weblogic.xml	/src/META-INF folder relative to the web application workspace	<ul style="list-style-type: none"> ▪ Role mapping of the valid-users security role to the Oracle Platform Security principal, users.
jazn-data.xml	./src/META-INF folder relative to the web application workspace	<ul style="list-style-type: none"> ▪ Default jazn.com realm name for the XML file-based identity store.

Specifically, when you select **ADF Authentication and Authorization** in the wizard, it sets the authorizationEnforce parameter in the <JaasSecurityContext> element to true. This allows the ADF security context to get the user principals from the HttpServletRequest once the user is authenticated by the container. For example, if a user clicks a link to a protected page and this user is not authenticated, the ADF authentication servlet is called and the web container invokes the login page. The user submits a user name and password and that data is compared against the data in the identity store where user information is stored. If a match is found, the originator of the request (the user) is authenticated. The user name is then stored in the ADF security context, where it can be accessed to obtain other security-related information (such as the group the user belongs to) in order to determine authorization rights.

You can proceed to define application roles and ADF security policies. The ADF security policy consists of a permission grant applied to the ADF resource and

assigned to the desired application role. Until you complete this step, all ADF resources are considered private and inaccessible even to authenticated users. Application roles are stored in the policy store and have no relationship to the identity store used to specify users and enterprise roles.

Note that you use different editors to define application roles and ADF security policies. You open both editors from the `jazn-data.xml` file, which appears in the **Descriptors/META-INF** node of the Application Resources panel:

- To define the security policies, you open the overview editor for the `jazn-data.xml` file by double-clicking the file.
- To define application roles, you open the Edit JPS Identity and Policy Store editor for the `jazn-data.xml` file by right-clicking the file and choosing **Edit**.

28.2.6 How to Disable ADF Security

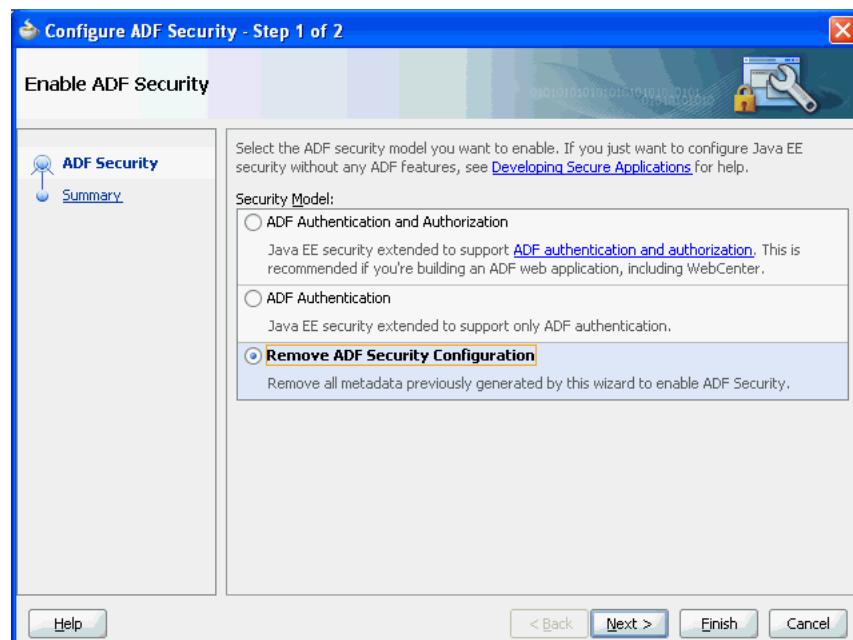
If you want to run the application without using Oracle ADF Security, open the Configure ADF Security wizard, select **Remove ADF Security Configuration** on the first page and all security will be turned off.

To disable ADF Security for the application:

1. In the Application Navigator, select the user interface project that contains the pages you want to secure. Do not select the model project or any other project.
2. From the **Tools** menu, choose **Configure ADF Security**.
3. In the ADF Security page, select **Remove ADF Security Configuration**, as shown in [Figure 28–7](#).

This selection will undo all ADF security-related configuration definitions from the `web.xml` file and the `adf-config.xml` file.

Figure 28–7 Disabling ADF Security in the Configure ADF Security Wizard



4. Click **Next** and then click **Finish**. All other wizard pages

28.2.7 What Happens When You Disable ADF Security

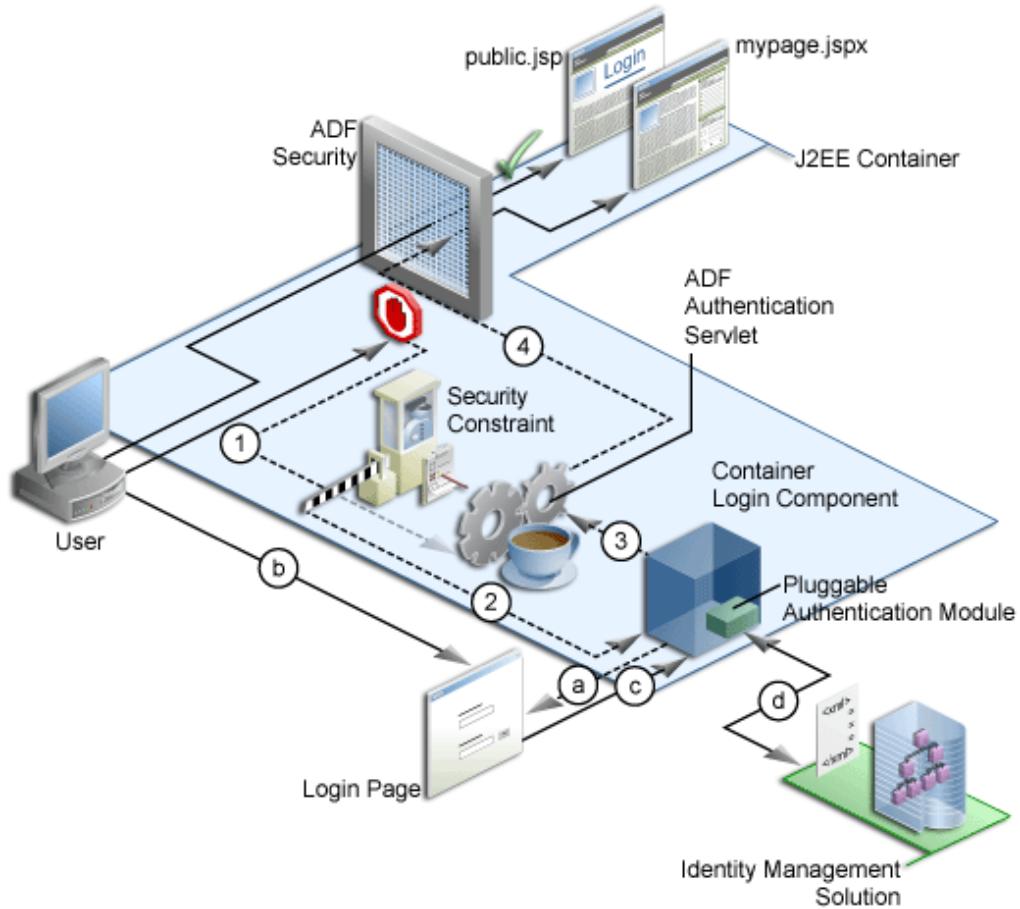
[Table 28–3](#) shows which files the Configure ADF Security wizard updates.

Table 28–3 Files Updated When Disabling ADF Security

File	File Location	Configuration Performed by the Configure ADF Security Wizard
web.xml	/public_html/WEB-INF folder relative to the user interface project	<ul style="list-style-type: none"> ■ Removes the Oracle adfAuthentication servlet definition. ■ Removes servlet mapping for the ADF authentication servlet. ■ Removes the constraint on the ADF authentication servlet. ■ Removes the login configuration. ■ Removes security role valid-users.
adf-config.xml	/.adf/META-INF folder relative to the web application workspace	<ul style="list-style-type: none"> ■ Removes the JAAS security context. ■ Removes the authenticationRequired and authorizationEnforce parameters in the <JaasSecurityContext> element.
weblogic.xml	/src/META-INF folder relative to the web application workspace	<ul style="list-style-type: none"> ■ Security role mapping of valid-users principal.

28.2.8 What Happens at Runtime: How Oracle ADF Security Handles Authentication

[Figure 28–8](#) illustrates the authentication process when the user attempts to access any page containing ADF resources (such as mypage.jspx) without first logging in. Authentication is initiated implicitly because the user does not begin log in by clicking a login link on a public page. In the case of the secured page, no grants have been made to the anonymous user.

Figure 28–8 Oracle ADF Security Implicit Authentication

In [Figure 28–8](#), the implicit authentication process assumes the resource does not have a grant to the `anonymous-role` role, that the user is not already authenticated, and that the authentication method is form-based authentication. In this case, the process is as follows:

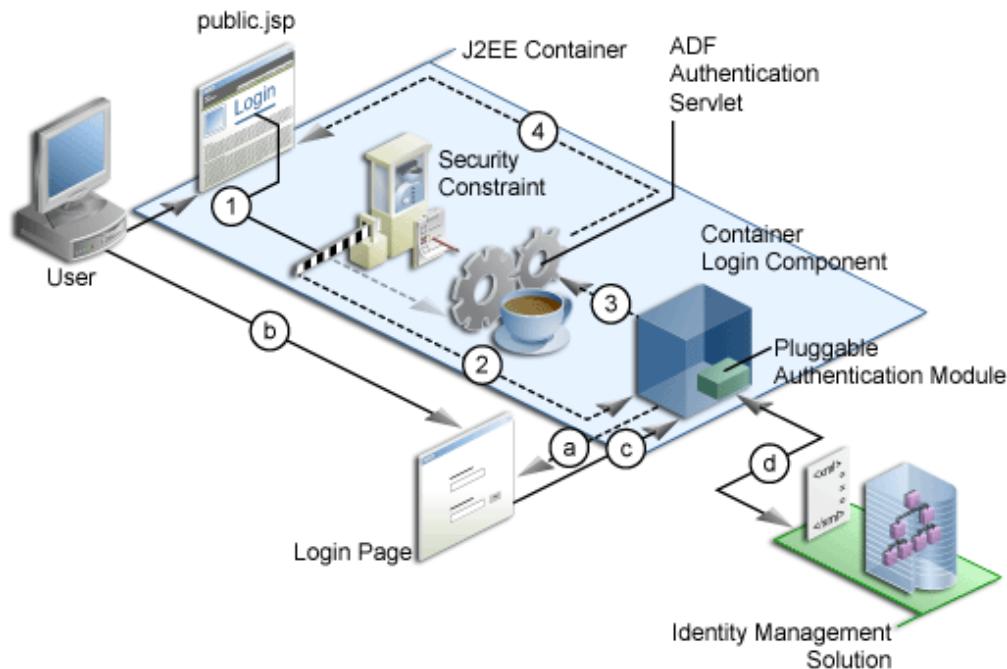
1. When the web page (or ADF task flow) is requested, the ADF bindings servlet filter redirects the request to the Oracle ADF authentication servlet (Step 1), storing the logical operation that triggered the login.
2. The ADF authentication servlet has a Java EE security constraint set on it, which results in the Java EE container invoking the configured login mechanism (Step 2). Based on the container's login configuration, the user is prompted to authenticate:
 1. The appropriate login form is displayed for form-based authentication (Step 2a).
 2. The user enters his credentials in the displayed login form (Step 2b).
 3. The user posts the form back to the container's `j_security_check()` method (Step 2c).
 4. The Java EE container authenticates the user, using the configured pluggable authentication module (Step 2d).
3. Upon successful authentication, the container redirects the user back to the servlet that initiated the authentication challenge, in this case, the ADF authentication servlet (Step 3).

4. On returning to the ADF authentication servlet, the servlet subsequently redirects to the originally requested resource (Step 4).

Whether or not the resource is displayed will depend on the user's access rights and whether authorization for ADF Security is enforced, as explained in [Section 28.2.9, "What Happens at Runtime: How Oracle ADF Security Handles Authorization"](#).

[Figure 28–9](#) illustrates the explicit authentication process when the user becomes authenticated starting with the login link on a public page.

Figure 28–9 Oracle ADF Security Explicit Authentication



In an explicit authentication scenario, an unauthenticated user (with only the anonymous user principal and anonymous-role principal) clicks the **Login** link on a public page (Step 1). The login link is a direct request to the ADF authentication servlet, which is secured through a Java EE security constraint in the `web.xml` file.

In this scenario, the current page is passed as a parameter to the ADF authentication servlet. As with the implicit case, the security constraint redirects the user to the login page (Step 2). After the container authenticates the user, as described in Step a through Step d in the implicit authentication case, the request is returned to the ADF authentication servlet (Step 3), which subsequently returns the user to the public page, but now with new user and role principals in place.

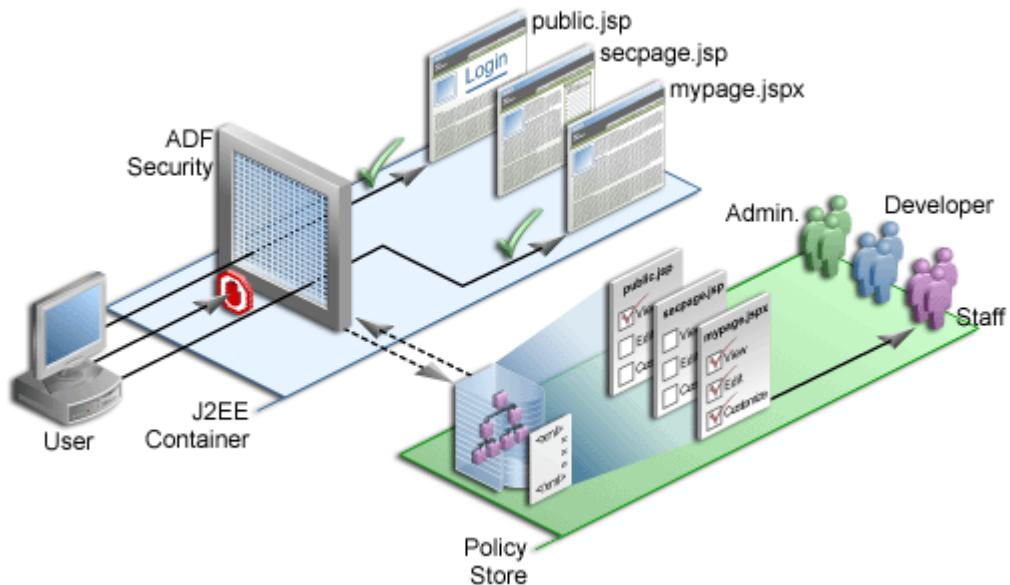
28.2.9 What Happens at Runtime: How Oracle ADF Security Handles Authorization

When ADF authorization is enabled, web pages that comprise an ADF task flow or any web page outside of a task flow that has an ADF page definition will be secure by default. When a user attempts to access these web pages, ADF Security checks to determine whether the user has been granted access in the policy store. If the user is not yet authenticated, and the page is not granted to the anonymous-role, then the application displays the login page or form. If the user has been authenticated, but does not have permission, a security error is displayed. If you do not configure the

policy store with appropriate grants, the pages will remain protected and therefore stay unavailable to the authenticated user.

Figure 28–10 illustrates the authorization process.

Figure 28–10 Oracle ADF Security Authorization



The user is a member of the application role *Staff* defined in the policy store. Because the user has not yet logged in, the security context does not have a Subject (a container object that represents the user). Instead, Oracle Platform Security provides Oracle ADF Security with a Subject with the anonymous user principal (a unique definition of the user) and the anonymous-role principal.

With the anonymous-role principal, typically the user would be able to access only pages not defined by ADF resources, such as the `public.jsp` page, whereas all pages that are defined either by an ADF task flow or outside of a task flow using an ADF page definition file are secure by default and unavailable to the user. An exception to security policy would be if you were to grant the anonymous-role role access to ADF resources in the policy store. In this case, the user would not be allowed immediate access to the page defined by an ADF resource.

When the user tries to access a web page defined by an ADF resource, such as `mypage.jspx` (which is specified by an ADF page definition, for example), the Oracle ADF Security enforcement logic intercepts the request and because all ADF resources are secured by default, the user is automatically challenged to authenticate (assuming that the anonymous-role is not granted access to the ADF resource, as previously mentioned).

After successful authentication, the user will have a specific Subject. The security enforcement logic now checks the policy store to determine which role is allowed to view `mypage.jspx` and whether the user is a member of that role. In this example for `mypage.jspx`, the View privilege has been granted to the Staff role and because the user is a member of this role, they are allowed to navigate to `mypage.jspx`.

Similarly, when the user tries to access `secpage.jsp`, another page defined by ADF resources, for which the user does not have the necessary View privilege, access is denied.

Users and roles are those already defined in the identity store of the resource provider. Application roles are defined in the policy store of the `jazn-data.xml` file.

28.3 Defining Users and Roles for a Fusion Web Application

Because web application security is based on a role-based access control mechanism with permissions granted to application roles, you must define a set of roles in the policy store that are specific to your application. For example, in the context of workflow, there may be roles such as customer, product specialist, supervisor, and administrator.

You will eventually map the application roles of your policy store to enterprise roles defined in the deployment environment. Mapping application roles will allow a user who is a member of a given enterprise role to access resources that are accessible from the associated application role. Enterprise roles are defined in the identity store of the security provider, such as `system-jazn-data.xml`, and are controlled only at an administrator level.

In the development environment, application roles exist in the `jazn-data.xml` file and are defined in `<app-role>` elements under `<policy-store>`. The location of the application policy store is

`application-root\src\META-INF\jazn-data.xml`. Application roles are defined to represent the policy requirements of the application, regardless of what may eventually be mapped to them from the identity store. The use of application roles allows development to proceed against functional roles, as described in [Section 28.3.2, "How to Define Application Roles in JDeveloper"](#).

Note: Application roles differ from roles that appear in the identity store portion of the `system-jazn-data.xml` file (defined in `<role>` elements under `<jazn-realm>`) or in roles defined by the enterprise LDAP provider. Application roles are specific to an application and defined in the application policy store. They are used by the application directly and are not necessarily known to Oracle WebLogic Server.

Because you will want to run your application in JDeveloper using Integrated WLS to test security, the Configure ADF Security wizard lets you generate the `test-all` application role, as described in [Section 28.3.1, "How to Enable the test-all Application Role in JDeveloper"](#). You can seed the identity store with users and then associate these test users with members of the `test-all` application role. Seeding the identity store with test users allows you to grant access rights to a few application roles and simulate the access rights of the application's actual users in a production environment. You edit the policy store in the `jazn-data.xml` file to associate the roles and their members, as described in [Section 28.3.3, "How to Configure the Identity Store with Test Users in JDeveloper"](#).

28.3.1 How to Enable the test-all Application Role in JDeveloper

When you run the Configure ADF Security wizard, you can add the `test-all` application role to the policy store in the `jazn-data.xml` file. When you enable this option in the wizard, you specify how you want grants to be made to the application role for ADF resources. The wizard lets you configure Oracle ADF Security so that JDeveloper will automatically add a View privilege to the policy store each time you or another application developer working on the user interface project creates a new ADF

resource. Alternatively, you can choose to limit the grant to the just the existing ADF resources of the user interface project.

After you run the wizard and enable the `test-all` role, you can later rerun the wizard and disable automatic grants at any time. Once disabled, new ADF task flows and web pages that you create will not utilize the `test-all` role and will therefore require that you define the ADF security policy or these resources will remain inaccessible to your testers or end-users, as described in [Section 28.4, "Defining ADF Security Access Policies"](#).

To configure ADF Security to use the test-all application role:

1. In the Application Navigator, select the user interface project that contains the pages you want to secure. Do not select the data model project or any other project.
2. From the Tools menu, choose **Configure ADF Security**.

For information about running the wizard, see [Section 28.2, "Choosing ADF Security Authentication and Authorization"](#).

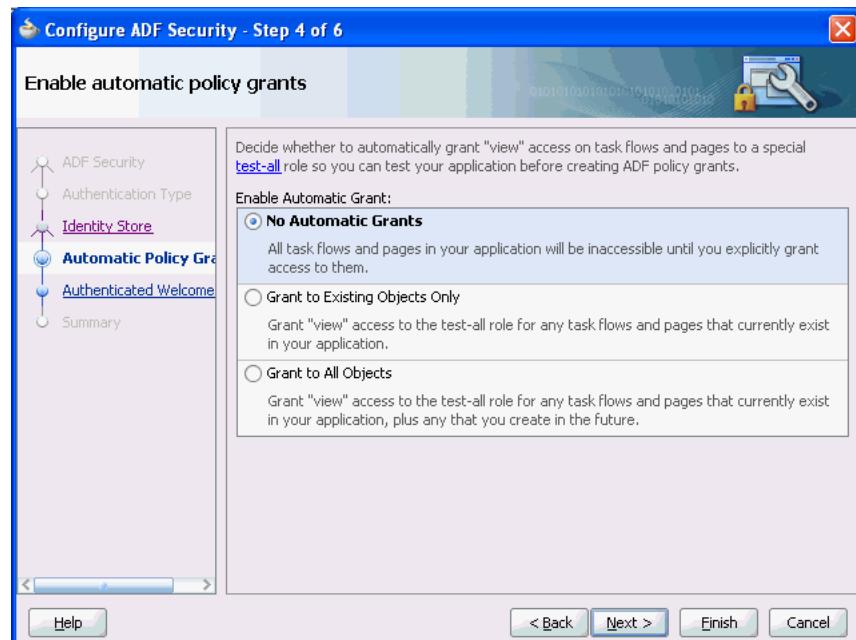
3. In the Configure ADF Security wizard, select **ADF Authentication and Authorization** in the ADF Security page.

If you select one of the other two options, the wizard disables the Automatic Policy Grants page in the wizard. Grants, whether automatic or explicit ones, will be checked by ADF resources only when ADF authorization is enforced.

4. In the next two pages, select the desired options and click **Next** until the wizard displays the Automatic Policy Grants page with the **No Automatic Grants** option selected by default, as shown in [Figure 28-11](#).

The default option for this page requires you to issue grants to either a developer-defined application role or to a built-in role (`anonymous-role` and `authenticated-role`) for all ADF resources.

Figure 28-11 The Configure ADF Security Wizard Disables the test-all Role By Default



5. In the Automatic Policy Grants page, select one of the two options that will enable automatic View grants to ADF resources, as follows:

Select **Grant to Existing Objects Only** when you want JDeveloper to grant View privileges to the `test-all` application role and you want this policy to apply to all your application's existing ADF task flows and web pages in the user interface project.

Select **Grant to All Objects** when you want JDeveloper to grant View privileges to the `test-all` application role and you want this policy to apply to all ADF task flows and web pages that developers create in the user interface project. Note that the wizard displays the option **Grant to New Objects** after you run the wizard the first time with the **Grant to All Objects** option selected.

6. Complete the remaining pages of the wizard and click **Finish**.

The `test-all` role appears in the `jazn-data.xml` file.

28.3.2 How to Define Application Roles in JDeveloper

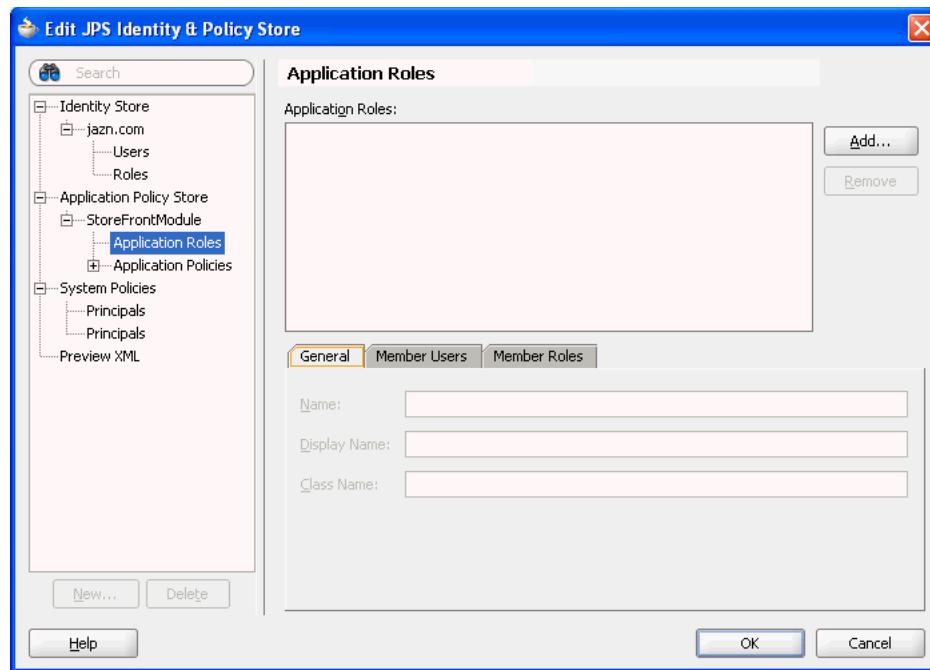
The policy store in ADF Security defines grants that establish access rights (or permissions) for specific ADF resources. These access rights are conferred on the authenticated user at runtime through the application role for which the user is defined as a member. Thus, before you can define access policies for ADF resources, you must specify one or more application roles in the `jazn-data.xml` file. During deployment these application roles will be associated with the enterprise roles that exist. Application roles are specific to the application and therefore provide a level of abstraction to the enterprise roles. You can add application roles to the policy store to distinguish as many levels of security as desired. Each application role will ultimately be associated with a specific set of grants to ADF resources, as described in [Section 28.4, "Defining ADF Security Access Policies"](#).

JDeveloper lets you edit the policy store to define application roles using either an overview editor for the ADF security policies or a property editor dialog. You access both editors from the context menu available on the `jazn-data.xml` file. When you work with property editor dialog, be sure to add the roles to the policy store and not to the identity store (you can also use the property editor to add enterprise security roles to the identity store that the XML security provider defines).

After you have completed the procedure to add an application role, you must associate members with that role, as described in [Section 28.3.3, "How to Configure the Identity Store with Test Users in JDeveloper"](#).

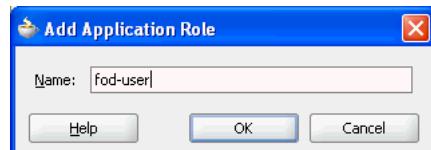
To add an application role to the policy store using the property editor dialog:

1. In the Application Navigator, right-click `jazn-data.xml` and choose **Properties** to display the property editor dialog for the XML-based security provider.
2. In the Edit JPS Identity and Policy Store dialog, create the policy store if one does not yet exist. Click **Application Policy Store** and click **New**.
If the `jazn-data.xml` file already has application roles defined, then you can add the new role to the existing application policy store, as described in Step 4.
3. In the Create Application Policy dialog, enter the display name for the policy store and click **OK**.
4. In the Edit JPS Identity and Policy Store dialog, expand the application policy store you created and select **Application Roles**, as shown in [Figure 28–12](#).

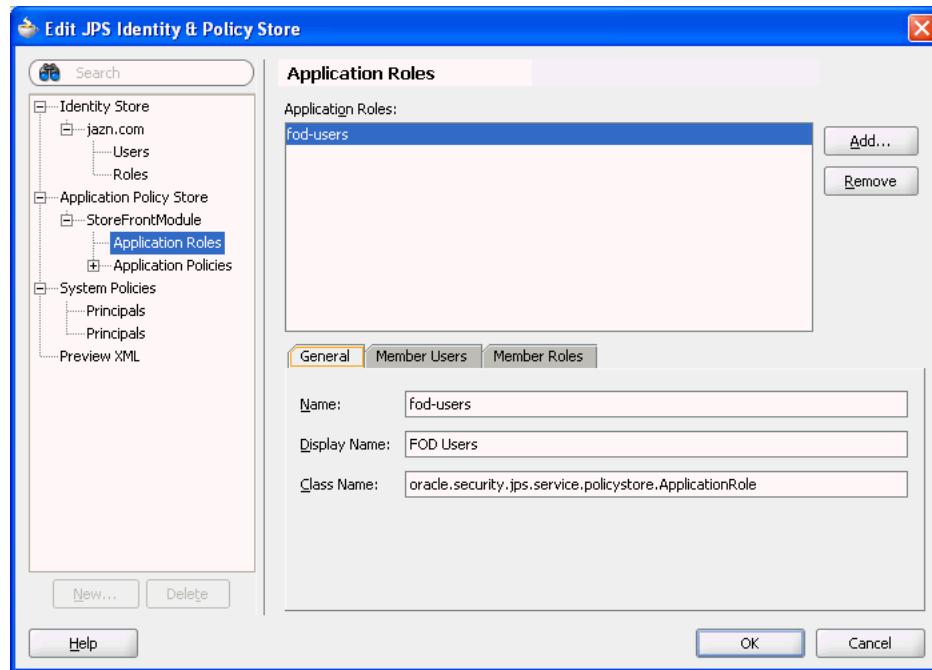
Figure 28–12 Editing the Policy Store Using the Property Editor

5. In the Application Roles page, click **Add** to create the application role.
6. In the Add Application Role dialog, enter the name of the role to add to the policy store.

For example, you might add the application role `fod-users`, as shown in [Figure 28–13](#).

Figure 28–13 Creating the Application Role

7. Click **OK** to redisplay the Edit JPS Identity and Policy Store dialog with the new application role, as shown in [Figure 28–14](#).

Figure 28–14 Displaying the New Application Role in the Property Editor

8. In the Edit JPS Identity and Policy Store dialog, click **OK** to add the application role to the policy store.

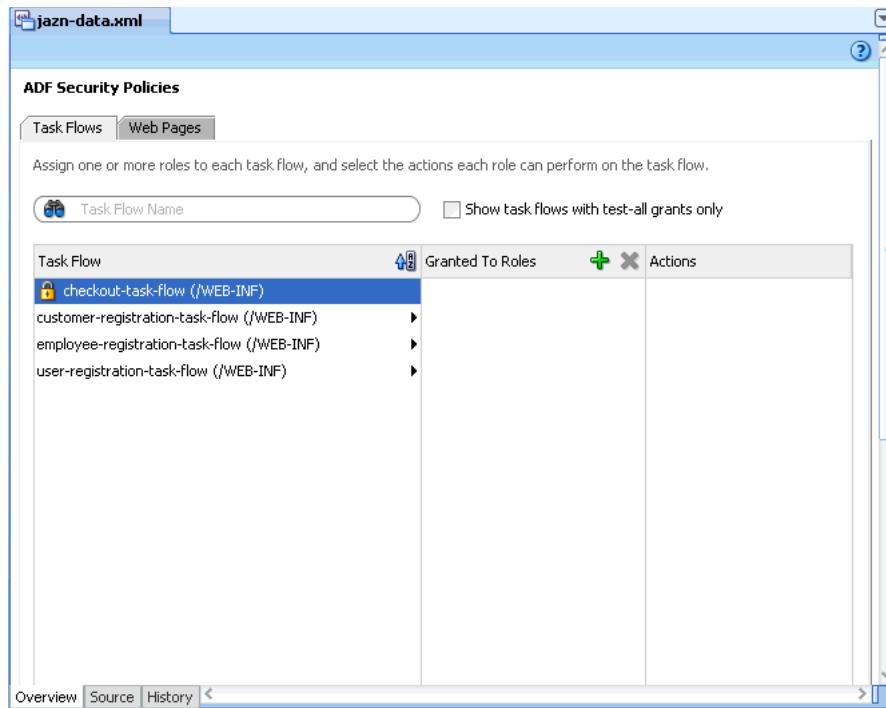
If you examine the source code for the `jazn-data.xml` file, you will see the application role entry as follows:

```
<app-role>
  <name>fod-users</name>
  <display-name/>
  <description/>
  <guid/>
  <class>oracle.security.jps.service.policystore.ApplicationRole</class>
</app-role>
```

To add an application role to the policy store using the overview editor:

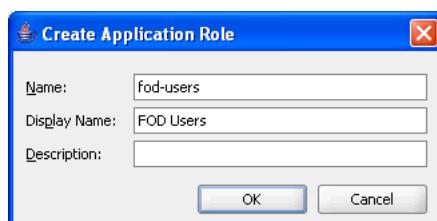
1. In the Application Navigator, right-click `jazn-data.xml` and choose **Open** to display the overview editor for ADF security policies.
2. In the overview editor, if the ADF resource is a task flow, select the task flow that you want to make a grant for and click the **Add Application Role** icon in the **Granted To Roles** column, as shown in [Figure 28–15](#).

When you just want to add an application role to the policy store, you can select any ADF resource displayed in the overview editor.

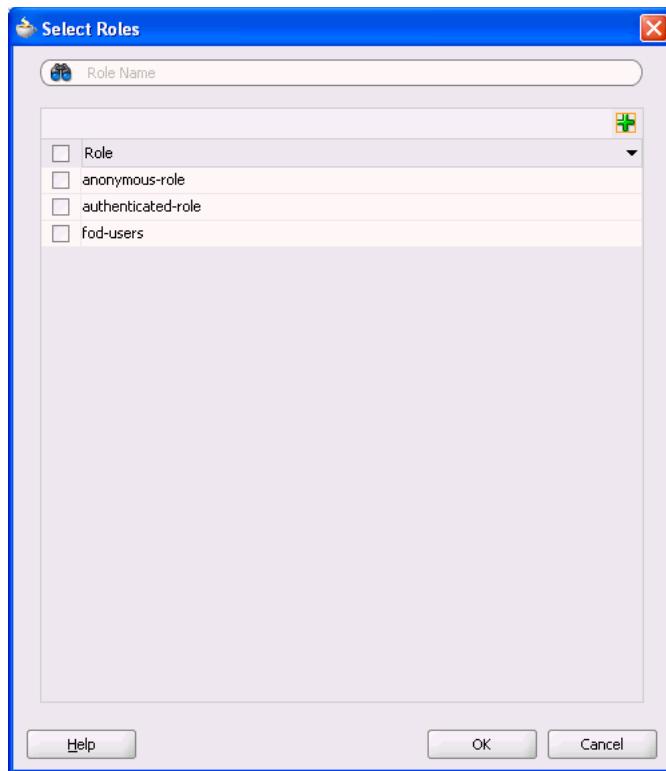
Figure 28–15 Editing the Policy Store Using the Overview Editor

3. In the Select Role dialog, click the **Add Application Role** button to create a new application role.
4. In the Create Application Role dialog, enter the name of the role that will be added to the policy store. Optionally, enter a display name and description.

For example, you might add the application role `fod-users`, as shown in [Figure 28–16](#).

Figure 28–16 Creating the Application Role

5. Click **OK** to redisplay the Select Roles dialog with the new application role, as shown in [Figure 28–17](#).

Figure 28–17 *Displaying the New Application Role in Select Roles Dialog*

6. In the Select Roles dialog, click **OK** to add the application role to the policy store.

If you examine the source code for the `jazn-data.xml` file, you will see an application role definition like this:

```
<app-role>
  <name>fod-user</name>
  <display-name/>
  <description/>
  <guid/>
  <class>oracle.security.jps.service.policystore.ApplicationRole</class>
</app-role>
```

28.3.3 How to Configure the Identity Store with Test Users in JDeveloper

You may want to seed the identity store with a temporary set of users to mirror the actual users' experience in your production environment. These test users can log in to the application and be conferred access rights to view the secure ADF resources of your application. To enable the user to view resources, you make grants against application roles rather than the users who are the members of those roles. Therefore, after you seed the identity store with test users, you must associate each user with an application role or they will not have sufficient privileges to view ADF resources.

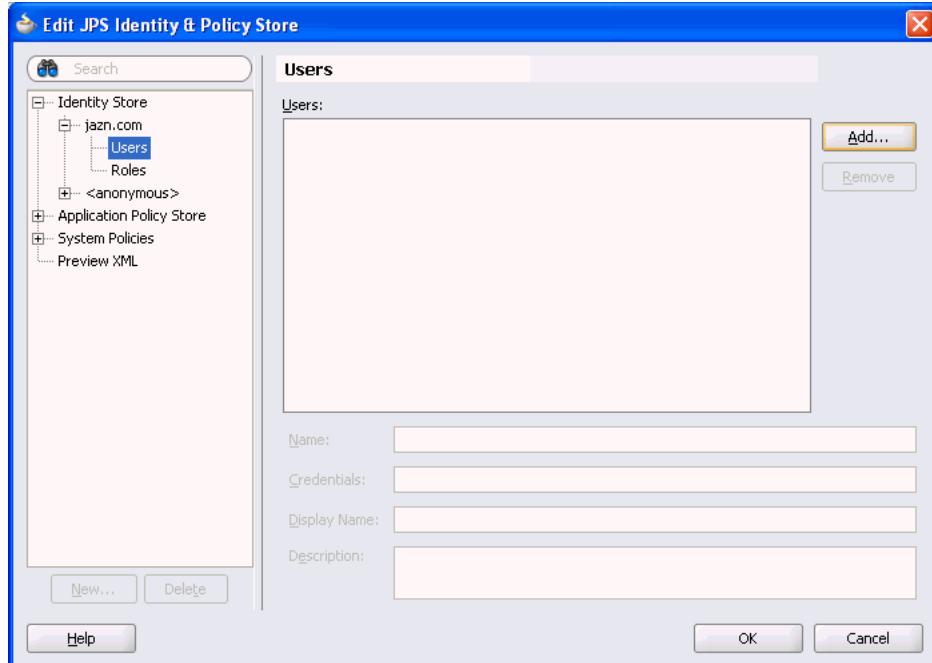
As a convenience when you use the Configure ADF Security wizard to add the `test-all` role to the policy store and you only require test users to be authenticated before they are allowed to access the ADF resources, you can add the test user accounts to the identity store and add the built-in application role `authenticated-role` to the `test-all` role. All users will automatically have the `authenticated-role` role once they log in.

You can add test users to application roles directly or as members of the built-in authenticated-role role.

To associate a test user with an application role using the test-all role:

1. In the Application Navigator, right-click **jazn-data.xml** and choose **Properties** to display the property editor dialog for the XML-based security provider.
2. In the Edit JPS Identity and Policy Store dialog, expand the **jazn.com** realm and click **Users** to view the list of users, as shown in [Figure 28–18](#).

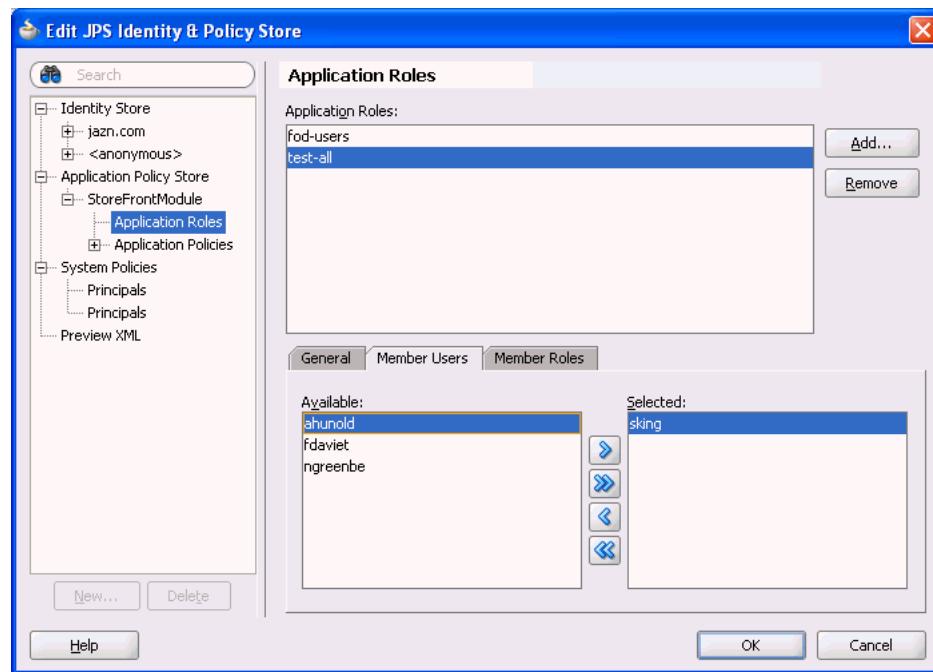
Figure 28–18 Editing the Identity Store Users Using the Property Editor



3. With the Users page displayed, click **Add** to create the test user.
4. In the Add User dialog, enter the login name and credentials that the user will be required to enter to become authenticated and click **OK**.
5. In the Edit JPS Identity and Policy Store dialog, expand the application policy store and select **Application Roles**.
6. In the Application Roles page, select the **test-all** role to define the members of this role.

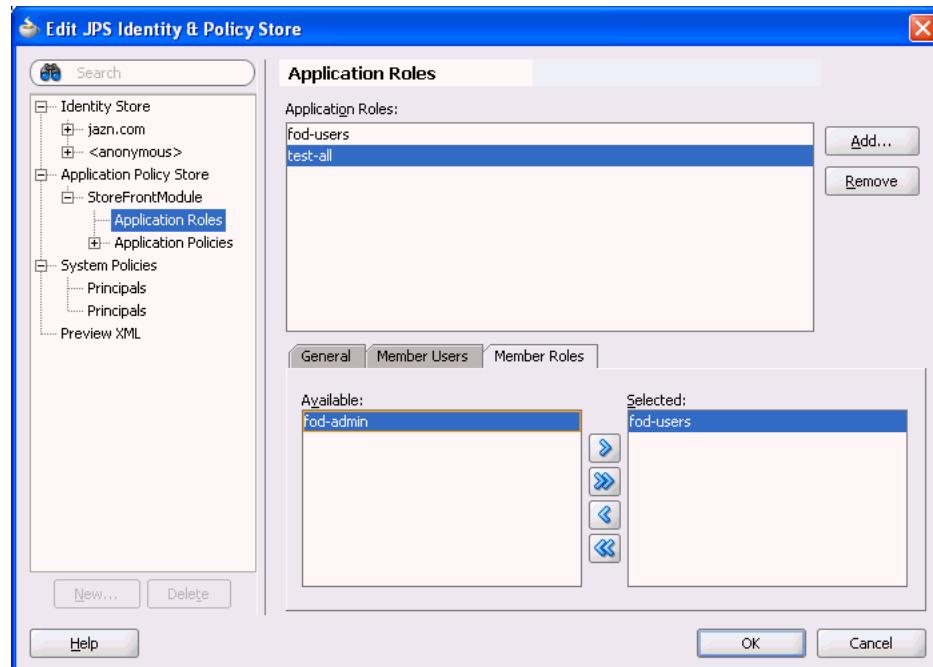
If the **test-all** role does not display in the Application Roles page, you need to configure Oracle ADF Security, as described in [Section 28.3.1, "How to Enable the test-all Application Role in JDeveloper"](#).

7. In the Application Roles page, click the **Member Users** tab and shuttle the desired test user to the **Selected** list, as shown in [Figure 28–19](#).

Figure 28–19 Adding the User to the test-all Role Using the Property Editor

8. Alternatively, you can add a system role that you created in the identity store to the **test-all** role. Click the **Member Roles** tab and shuttle the desired system role to the **Selected** list, as shown in [Figure 28–20](#).

All authenticated members of the system role will have access rights to the ADF resources with a View privilege grant to the **test-all** role.

Figure 28–20 Adding a System Role to the test-all Role Using the Property Editor

9. In the Edit JPS Identity and Policy Store dialog, click **OK** to update the `test-all` role in the policy store.

If you examine the source code for the `jazn-data.xml` file, you will see the `test-all` role definition with member entries like this:

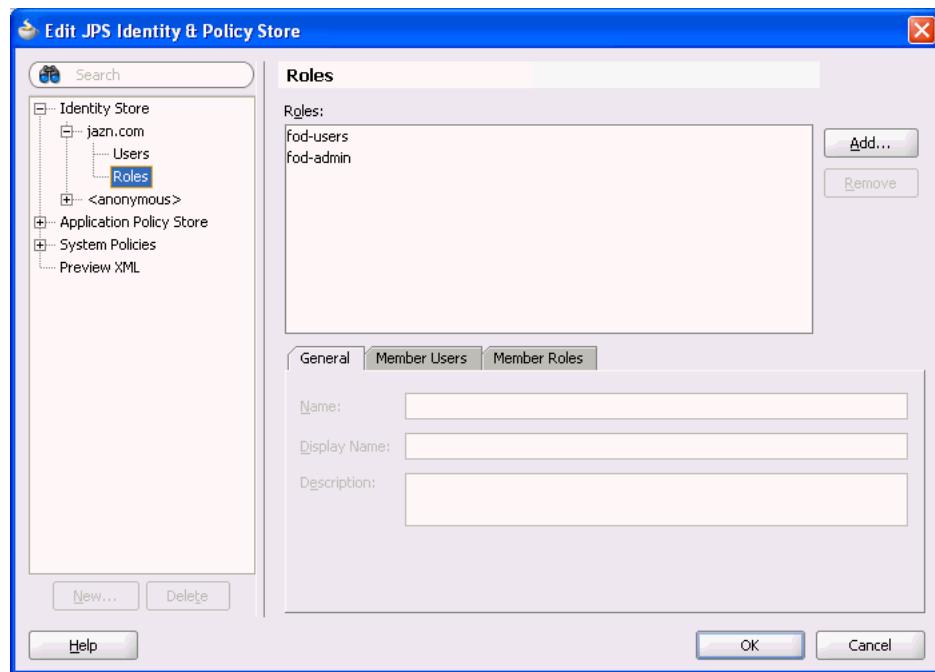
```
<app-role>
  <name>test-all</name>
  <guid>FFFFF394F68</guid>
  <display-name>test-all</display-name>
  <description/>
  <class>oracle.security.jps.service.policystore.ApplicationRole</class>
  <members>
    <member>
      <name>sking</name>
      <class>weblogic.security.principal.WLSGroupImpl</class>
    </member>
    <member>
      <name>fod-users</name>
      <class>oracle.security.jps.internal.core.principals.
          JpsXmlEnterpriseRoleImpl</class>
    </member>
  </members>
</app-role>
```

You can ensure that any authenticated user will have the access rights granted to the `test-all` application role. You accomplish this by making the built-in `authenticated-role` role a member of the `test-all` application role.

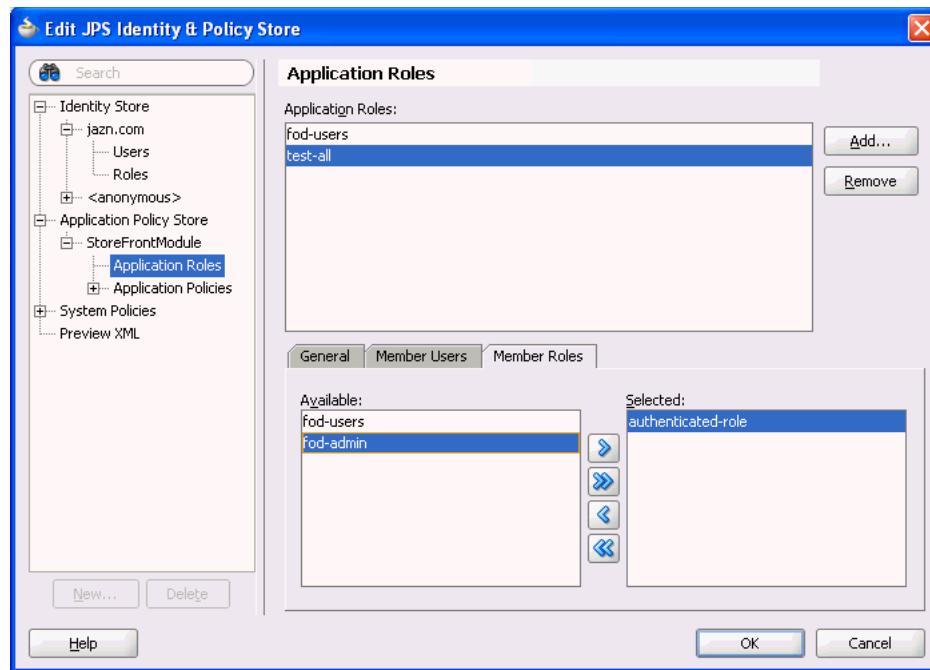
To associate any authenticated user with the test-all role:

1. In the Application Navigator, right-click `jazn-data.xml` and choose **Properties** to display the property editor dialog for the XML-based security provider.
2. In the Edit JPS Identity and Policy Store dialog, expand the `jazn.com` realm and click **Roles** to view the list of system roles, as shown in [Figure 28-21](#).

Figure 28–21 Editing the Identity Store Roles Using the Property Editor



3. With the Roles page displayed, click **Add** to create the authenticated-role system role.
4. In the Add Role dialog, enter the built-in role **authenticated-role** and click **OK**.
5. In the Edit JPS Identity and Policy Store dialog, expand the application policy store and select **Application Roles**.
6. In the Application Roles page, click the **Member Roles** tab and shuttle the **authenticated-role** role to the **Selected** list, as shown in [Figure 28–22](#).

Figure 28–22 Adding the authenticated-role to the test-all Role Using the Property Editor

- In the Edit JPS Identity and Policy Store dialog, click **OK** to update the `test-all` role in the policy store.

If you examine the source code for the `jazn-data.xml` file, you will see the `test-all` role definition with an `authenticated-role` member entry like this:

```
<app-role>
  <name>test-all</name>
  <display-name>test-all</display-name>
  <description/>
  <guid/>
  <class>oracle.security.jps.service.policystore.ApplicationRole</class>
  <members>
    <member>
      <name>authenticated-role</name>
      <class>oracle.security.jps.internal.core.principals.
          JpsXmlEnterpriseRoleImpl</class>
    </member>
  </members>
</app-role>
```

28.4 Defining ADF Security Access Policies

Authorization relies on a policy store definition that is accessed at runtime and that contains permissions that grant privileges to execute predefined actions, like `view`, on a specified object. Initially, after you run the Configure ADF Security wizard, the policy store defines no grants. Because ADF resources are secure by default, they will be unavailable to users. You must use JDeveloper to define grants for the resources that you want to permit users to access.

Oracle ADF Security defines the following authorization points for your application's access policies:

- Groups of web pages with grants issued on ADF task flows.

In this case, the web pages that the task flow defines will all be securable with grants.

- Individual web pages with grants issued on ADF page definitions. This is useful for any web page that is not contained in an ADF task flow.

Typically, the page definition contains a set of ADF binding definitions used to databind ADF Faces components. However, you can create an empty page definition file on any web page when you want to secure the page that does not display databound components.

- Specific row-level data accessed through the Business Components data model project, with grants issued on entity objects or their attributes.

Before you can define security policies, your policy store for your application must contain the application roles that you intend to issue grants to. This can be an application role that you define (such as `fod-users`) or it can be one of the two built-in application roles: `authenticated-role` or `anonymous-role`. You can use these application roles to classify users, so that each member of the same role possess the same access rights. As such, the security policy names the application role as the principal of the grant, rather than specific users. For details about defining application roles, see [Section 28.3, "Defining Users and Roles for a Fusion Web Application"](#).

Tip: If you are not ready to define application roles for the users of your application, consider enabling the `test-all` role in the Configure ADF Security wizard, as described in [Section 28.3.1, "How to Enable the test-all Application Role in JDeveloper"](#).

For the view-controller project, you use the overview editor for ADF security policies to secure ADF resources, including ADF task flows and ADF page definitions. In JDeveloper, you open the editor on the `jazn-data.xml` file from the Application Resources panel in JDeveloper. Note that you can edit the file using two different editors that you select from the file's context menu:

- If you right-click the file and choose **Open** (or just double-click the file), you will get the overview editor for the ADF security policies. This is the central policy editor in JDeveloper. You can use it to define set security policies for all ADF resources.
- If you right-click the file and choose **Properties**, you will open the property editor Edit JPS Identity and Policy Store dialog. This is the property editor that you can use to add users, roles, and application roles. It is not as convenient as the overview editor for granting permissions on the application roles.

ADF Security relies on the `jazn-data.xml` file for the policy store whether you are using the XML-based identity store or the LDAP identity store. Thus, with Oracle ADF Security, you define user interface access policies in two steps:

1. Define an application role for which you will make the ADF resource grant.

For details about defining application roles, see [Section 28.3.2, "How to Define Application Roles in JDeveloper"](#).

2. Select actions that you want to grant on the Permission that secures the ADF resource.

Together these two steps allow you to define the access policy for authenticated users who belong to an application role with sufficient privileges to access the resource.

For details about granting permissions for row-level security, see [Section 28.4.3, "How to Secure Row Data Using ADF Business Components"](#).

28.4.1 How to Grant Permissions on ADF Bounded Task Flows

Certain Oracle ADF resources are security-aware, meaning predefined resource-specific permissions exist that a developer can grant. Among these resources are ADF task flow definitions that JDeveloper creates for you when you use the ADF task flow diagrammer to create a task flow. By defining an access policy for the task flow definition, you can control the user's ability to view the web pages of the entire task flow.

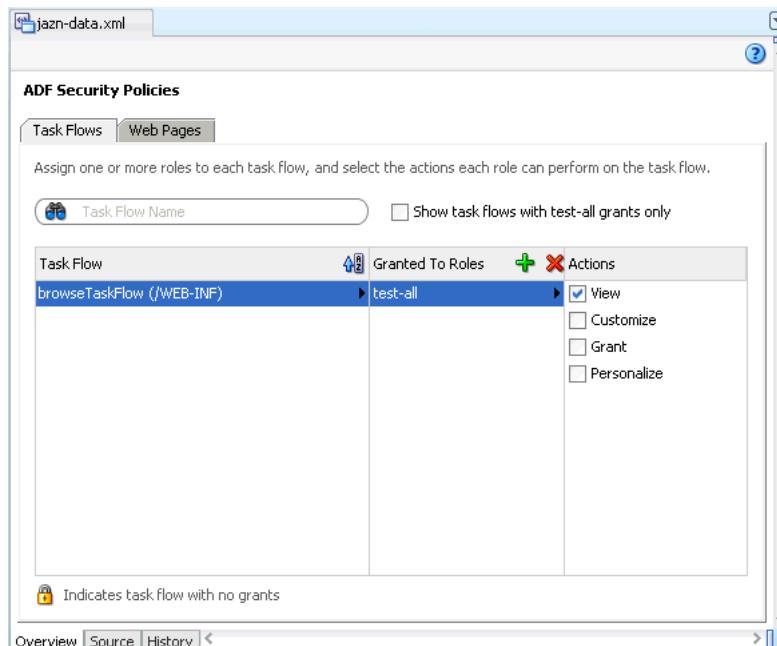
Objects within the bounded task flow inherit the task flow's access policy. For example, authorization to an ADF bounded task flow defaults authorization for any pages and other ADF bounded task flows contained within it. You cannot override the default authorizations at the task flow's page level. For cases where you need to issue grants to individual web pages that are not part of a task flow, see [Section 28.4.2, "How to Grant Permissions on Individual Web Pages Using ADF Page Definitions"](#).

The access policy represents a set of privileges required to perform a set of operations for a task flow. If authorization is given to view the task flow, the task flow can also be executed. If authorization is not given, a runtime exception will be received. The calling task flow can handle the exception within its Exception Handler activity.

ADF unbounded task flows are not securable.

You define the access policy for an ADF bounded task flow by selecting the task flow definition name in the overview editor for ADF security policies, as shown in [Figure 28–23](#). The selections you make will appear as metadata in the policy store section of the `jazn-data.xml` file. This metadata defines a permission target for which you have issued grants to authorize the members of a specific application role.

Figure 28–23 Grant Made to Task Flow in Overview Editor



The list of available actions displayed by the overview editor is defined by the task flow permission class (`oracle.adf.controller.security.TaskFlowPermission`). The permission class maps these actions to the operations supported by the task flow. [Table 28–4](#) shows the grantable actions of ADF bounded task flows.

Table 28–4 Secured Actions of ADF Bounded Task Flows

Grantable Action	Affect on the User Interface
View	Read and execute a bounded task flow. This is the only operation that the task flow supports in this release.

To define a grant for the task flow security policy, use the Task Flows page of the overview editor that you open for the `jazn-data.xml` file.

To define a permission grant on an ADF bounded task flow:

1. In the Application Resources panel, double-click the `jazn-data.xml` file located in the **Descriptors/META-INF** node.

2. In the overview editor, select the **Task Flows** tab.

The Task Flows page displays all the task flows that your application defines. Task flows are defined by `adfc-config.xml` files that appear in the Web Content/Page Flows node of the user interface project.

3. In the **Task Flows** column, select the task flow for which you want to grant access rights.

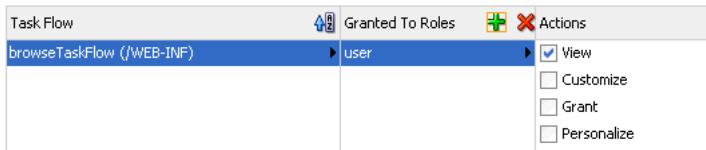
Use the search field to limit the task flows displayed in the overview editor.

4. In the **Granted to Roles** column, click the **Add Application Role** button and select the application role from the Select Role dialog that you want to make a grantee of the permission.

The Select Roles dialog displays application roles from the `jazn-data.xml` file. If the dialog displays no application roles, you must define at least one application role, as described in [Section 28.3.2, "How to Define Application Roles in JDeveloper"](#).

5. In the **Actions** column, select the action you want to grant for the role so that it appears enabled.

The `TaskFlowPermission` class defines task flow--specific actions that it maps to the task flow's operations, as described in [Table 28–4](#). For example, to grant **View** permission to the **user** role, select it as shown in [Figure 28–24](#).

Figure 28–24 Grant Made to the User Role for an ADF Task Flow Definition

6. You can repeat these steps to make additional grants as desired.

The same task flow definition can have multiple grants made for different application roles. The grants appear in the policy store definition of the `jazn-data.xml` file.

28.4.2 How to Grant Permissions on Individual Web Pages Using ADF Page Definitions

Certain Oracle ADF resources are security-aware, meaning predefined resource-specific permissions exist that a developer can grant. Among these resources

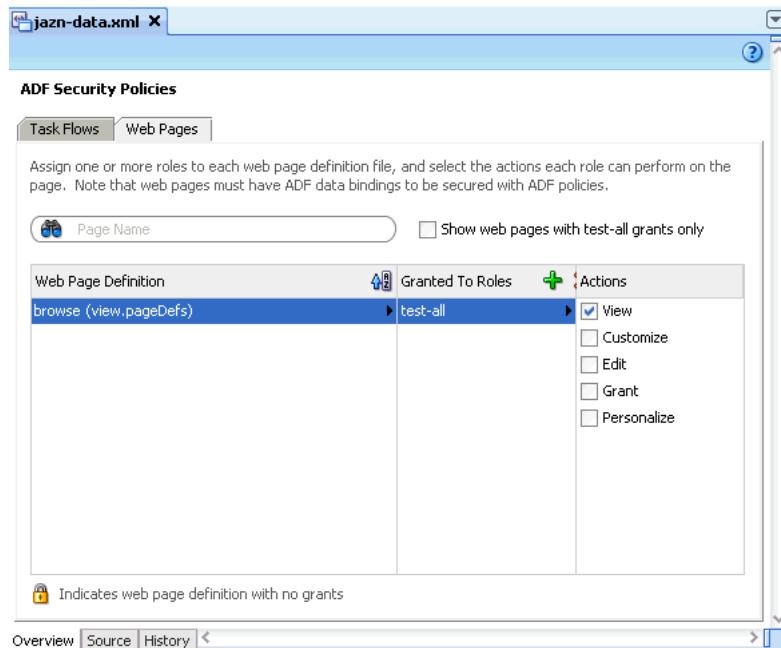
are ADF page definitions that JDeveloper creates for you when you work with the ADF data controls. By defining an access policy for the page definition, you can control the user's ability to view the web page itself. There is a one-to-one relationship between the page definition file and the web page it secures.

You can issue grants for an existing ADF page definition. JDeveloper creates the page definition file for you when you use ADF data controls to add ADF Faces components to the web page. Alternatively, when your web page does not need to use ADF data controls, for example to databind ADF Faces components, you can create the page definition file yourself by right-clicking the file and choosing **Go to Page Definition**.

There is a one-to-one relationship between the page definition file and the web page it secures.

You define the access policy for an ADF page definition by selecting the page definition name in the overview editor for ADF security policies, as shown in [Figure 28–25](#). The selections you make will appear as metadata in the policy store section of the `jazn-data.xml` file. This metadata defines a permission target for which you have issued grants to authorize the members of a specific application role.

Figure 28–25 Grant Made to test-all Role for an ADF Page Definition



The list of available actions displayed by the overview editor is defined by the region permission class (`oracle.adf.share.security.authorization.RegionPermission`). The permission class maps these actions to the operations supported by the ADF page definition for the web page. [Table 28–5](#) shows the grantable actions of the ADF page definition.

Table 28–5 Securable Actions of ADF Page Definitions

Grantable Action	Affect on the User Interface
View	<p>View the page.</p> <p>This is the only operation that the page definition supports in this release.</p>

To define a grant for the page definition security policy, use the Web Pages page of the overview editor that you open for the `jazn-data.xml` file.

To define a permission grant on an ADF page flow:

1. In the Application Resources panel, double-click the `jazn-data.xml` file located in the **Descriptors/META-INF** folder.
2. In the overview editor, select the **Web Pages** tab.

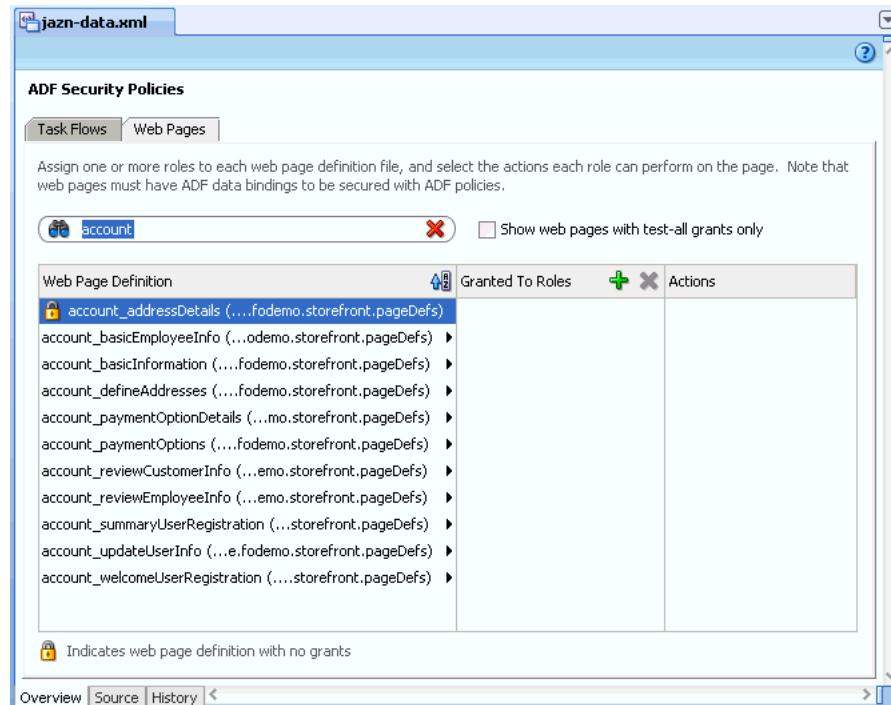
The Web Pages page of the overview editor displays all web pages that have an associated ADF page definition. This includes any web page that uses ADF data bindings or any web page for which you created an empty page definition. Page definitions are defined by `PageDef.xml` files that appear in the Application Sources node of the user interface project.

3. In the **Web Page Definition** column, select the web page and associated page definition for which you want to grant access rights.

The web page definition may display the lock symbol if you have not already made the page accessible to an application role. For example, the page `account_addressDetails.jsff` shown in [Figure 28–26](#) is still secure since no grant has been made.

Use the search field to limit the page definitions displayed in the overview editor. For example, a search on the word `account` would display only the web pages that begin with the word `account`, as shown in [Figure 28–26](#).

Figure 28–26 Limiting Web Pages Displayed in the Overview Editor



4. In the **Granted to Roles** column, click the **Add Application Role** icon and select the application role from the Select Roles dialog that you want to make a grantee of the permission.

The Select Roles dialog displays application roles from the `jazn-data.xml` file. You can define an application role or use one of the built-in application roles, as described in [Section 28.3.2, "How to Define Application Roles in JDeveloper"](#).

5. In the **Actions** column, select the action you want to grant for the role so that it appears enabled.

The `RegionPermission` class defines page definition--specific actions that it maps to the page's operations, as described in [Table 28–5](#). For example, to grant **View** permission to the **fod-users** role, select it as shown in [Figure 28–27](#).

Figure 28–27 Granting to the *fod-users* Role for an ADF Page Definition

Web Page Definition	Granted To Roles	Actions
account_addressDetails (...odemo.storefront.pageDefs)	fod-users	<input checked="" type="checkbox"/> View <input type="checkbox"/> Customize <input type="checkbox"/> Edit <input type="checkbox"/> Grant <input type="checkbox"/> Personalize
account_basicEmployeeInfo (...odemo.storefront.pageDefs)		
account_basicInformation (...fodemo.storefront.pageDefs)		
account_defineAddresses (...fodemo.storefront.pageDefs)		
account_paymentOptionDetails (...mo.storefront.pageDefs)		
account_paymentOptions (...fodemo.storefront.pageDefs)		
account_reviewCustomerInfo (...emo.storefront.pageDefs)		
account_reviewEmployeeInfo (...storefront.pageDefs)		
account_summaryUserRegistration (...storefront.pageDefs)		
account_updateUserInfo (...e.fodemo.storefront.pageDefs)		
account_welcomeUserRegistration (...storefront.pageDefs)		

6. You can repeat these steps to make additional grants as desired.

The same page definition can have multiple grants made for different application roles. The grants appear in the policy store definition of the `jazn-data.xml` file.

28.4.3 How to Secure Row Data Using ADF Business Components

Oracle ADF entity objects in the model project are security-aware, meaning predefined resource-specific permissions exist that a developer can grant. Additionally, you can secure just the individual attributes of entity objects.

Entity objects that you secure restrict users from updating data displayed by any web page that renders a UI component bound by an ADF binding to the data accessed by the secured entity object. Additionally, when you secure an entity object, you effectively secure any view object in the data model project that relies on that entity object. As such, entity objects that you secure define an even broader access policy that applies to all UI components bound to this set of view objects. For details about entity-based view objects, see [Section 5.2, "Populating View Object Rows from a Single Database Table"](#).

28.4.3.1 Defining a Permission on ADF Business Component Entity Objects

You can secure operations of the entity objects or their individual attributes.

In the data model project, you use the business component's overview editor to define a permission map for the specific actions allowed by the business component. The metadata consists of a permission class, permission name, and a set of actions mapped to binding operations.

The list of available operations displayed by the overview editor is defined by the entity object permission class (`oracle.adf.share.security.authorization.EntityPermission`). The

permission class maps the operations supported by the entity object to actions. Table 28–6 shows the securable operations of the entity object.

Table 28–6 Securable Operations of Oracle ADF Business Components

ADF Component	Securable Operation	Expected Mapped Action	Corresponding Implementation
ADF Business Components entity objects	read	Read	View the rows of a result set that has been restricted by a WHERE clause fragment.
	removeCurrentRow	Delete	Delete a row from the bound collection.
	update	Update	Update any attribute of the bound collection.
ADF Business Components attributes of entity objects	update	Update	Update a specific attribute of the bound collection.

To secure all row-level data that the entity object accesses, use the overview editor for the business component.

To secure an operation on an entity object:

1. In the data model project displayed in the Application Navigator, double-click the entity object that you want to secure.
2. In the General page of the overview editor, expand the **Security** section. The **Security** section displays the securable operations that the EntityPermission class defines. The class maps the entity object–specific actions to the entity object’s operations, as described in Table 28–6.
3. In the **Enabled** column, select the operations you want to secure for the entity object.

For example, to enable **read** permission, select it as shown in Figure 28–28.

Figure 28–28 Permission Enabled on read Operation for an ADF Entity Object

Security: Orders1	
Operation	Enabled
read	<input checked="" type="checkbox"/>
update	<input type="checkbox"/>
removeCurrentRow	<input type="checkbox"/>

The permissions appear in the XML definition of the entity object.

To secure individual columns of data that the entity object accesses, use the attribute page of the overview editor for the business component.

To secure an operation on an entity object attribute:

1. In the data model project displayed in the Application Navigator, double-click the entity object that defines the attribute you want to secure.
2. In the Attributes page of the overview editor, select the desired attribute, and expand the **Security** section.

The **Security** section displays the securable operations that the EntityAttributePermission class defines. The class maps the entity object-specific actions to the entity object's operations, as described in [Table 28–6](#).

3. In the **Enabled** column, select the operation you want to secure for the entity object attribute.

For example, to enable **update** permission, select it as shown in [Figure 28–29](#).

Figure 28–29 Permission Enabled on update Operation for an ADF Entity Object Attribute

Operation	Enabled
update	<input checked="" type="checkbox"/>

The permission map appears in the XML definition of the entity object.

28.4.3.2 Granting Permissions on ADF Business Components

Once a permission target is configured, any data that derives from entity objects or their attributes remain unsecured until you explicitly define access rights for the entity object's permission target.

To define the access policy for an existing business component permission target, use the Edit Authorization dialog.

To define the access policy for an entity object:

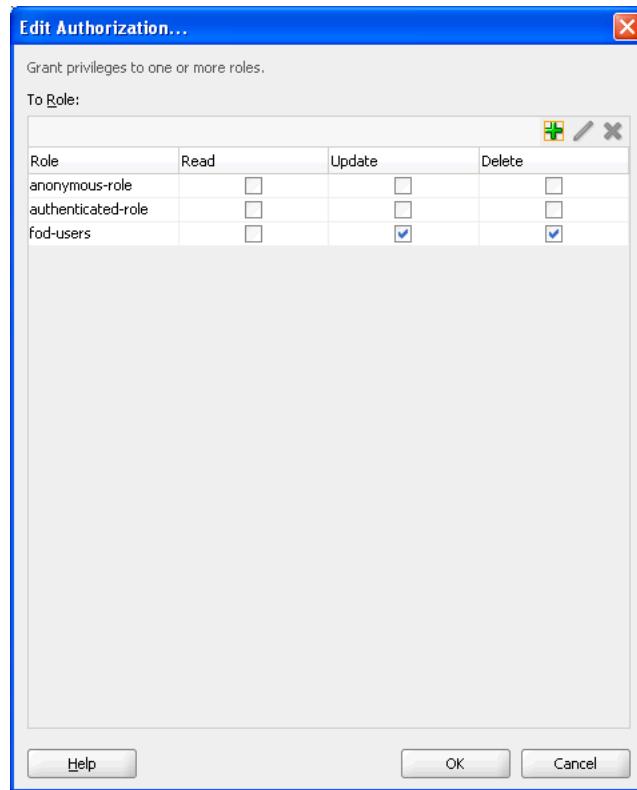
1. In the data model project displayed in the Application Navigator, locate the entity object and select it.
2. In the Structure window for the selected entity object or entity object attribute, right-click and choose **Edit Authorization**.

The Edit Authorization dialog displays the available actions of the entity object (or attribute), as described in [Table 28–6](#). The dialog also displays the application roles from the `jazn-data.xml` policy store. The built-in application roles **anonymous-role** and **authenticated-role** will appear with application roles that the application developer created.

3. In the dialog, select the action that you want to grant to a specific application role.

For example, to grant **Update** and **Delete** privileges to the **fod-users** application role, select those actions, as shown in [Figure 28–30](#).

The grant to the application role appears in the `jazn-data.xml` file.

Figure 28–30 Defining the Access Policy for an Entity Object

28.4.4 What Happens When You Define a Security Policy for ADF Resources

The overview editor for the `jazn-data.xml` file exposes application-level roles that are defined in the policy store (the `jazn-data.xml` file located in the `/src/META-INF` node relative to the web application workspace.) and displays the actions that are defined against a specific component type. To implement the security policy, you select the desired action against one or more of the displayed roles.

Note: In this release, policy information in the JAAS policy store is scoped by application. This scoping allows two applications to refer to the same permission target, without producing unintentional results. It is no longer necessary to name application resources to impose application scoping of the policy information.

The overview editor writes the permission information to the policy store. The policy defines a permission type, the resource that is secured, the actions that can be performed against that resource, and to whom that policy is being assigned or granted. The policy is defined by a grant, which contains both a grantee and one or more permissions.

A grantee defines to whom the policy is applied. [Example 28–1](#) shows how grants are defined in the `jazn-data.xml` file.

Example 28–1 Grants in the `jazn-data.xml` file

```
<jazn-policy>
  <grant>
    <grantee>
```

```

<principals>
  <principal>
    <class>oracle.security.jps.service.policystore.ApplicationRole</class>
    <name>fod-user</name>
  </principal>
</principals>
</grantee>
<permissions>
  <permission>
    <class>oracle.adf.controller.security.TaskFlowPermission</class>
    <name>/WEB-INF/checkout-task-flow.xml#checkout-task-flow</name>
    <actions>view</actions>
  </permission>
  <permissions>
    <permission>
      <class>oracle.adf.share.security.authorization.RegionPermission</class>
      <name>oracle.fodemo.storefront.pageDefs.homePageDef</name>
      <actions>view</actions>
    </permission>
    ...
  </permissions>
  ...
</grant>
...
<jazn-policy/>

```

In this grant, the role principal *fod-user* has been assigned *view* permission on the *checkout-task-flow* and the home web page with the *homePageDef* page definition. For the web page, notice that permission has been specified against the *homePageDef* page definition associated with the home page.

When authorization is enabled for Oracle ADF Security, JAAS authorization checks are performed on all ADF bounded task flows and all web pages defined by an ADF page definition.

The authorization check for a task flow determines whether the user has the required permission (typically, *view* permission). If the user is not authorized, an exception is thrown and ADF controller passes control to a designated exception handler. This designated exception handler should not be invoked inside the secured task flow, as this could result in a security violation. Instead, you can invoke the exception handler for secure task flows somewhere in the task flow's calling stack (see [Section 17.4, "Handling Exceptions"](#) for more information). If the user is authorized, then the task flow is entered.

28.4.5 What You May Need to Know About ADF Resource Grants

Grants that you make for ADF resources are standard JAAS Permissions. The web container will utilize the grants when ADF Security is enabled to allow authorization. In the authorization mode, ADF Security uses fine-grained authorization, implemented with JAAS Permissions to perform security checks for access rights to pages. The ADF Security enforcement logic checks to see whether the user, represented by the JAAS Subject, has the right permissions to access the resource.

The Subject contains the user's Principals, which include a user principal that contains their name (could be anonymous, before logging on, or some user name after logging on), and their list of role principals, which would include *anonymous-role* and some number of other roles that are obtained from the policy and identity stores. The Principal is created to represent all of the user's memberships in identity store roles and application roles in the policy store.

In turn, each of these roles may have multiple Permissions associated with them. These are the permission grants that are assigned through the overview editor you use on the `jazn-data.xml` file to indicate which roles have access to which pages. The grants are expected to be made against application roles and not directly to identity store roles.

During deployment, you will need to edit the application roles in the policy store to reflect who from the identity store belongs to each role. You can update the roles with members that are users or identity store roles.

Then at runtime, whether the current user has view permission on the page they are trying to access will be determined by the task flow controller or by the ADF Model when the user accesses a web page outside of a task flow and that page is associated with an ADF page definition. Oracle Security Platform then checks to see whether the Subject contains the roles that have the corresponding Permissions needed to access the page.

28.4.6 How to Use Regular Expressions to Define Policies on Groups of Resources

Consider [Example 28–2](#), in which each method name starts with `method_`.

Example 28–2 Method Permission Defined in the `jazn-data.xml` File

```
<principal>
  <class>oracle.security.jps.service.policystore.ApplicationRole</class>
  <name>anonymous-role</name>
</principal>

<permission>
  <class oracle.adf.share.security.authorization.MethodPermission </class>
  <name>method_1</name>
  <actions>invoke</actions>
</permission>

<permission>
  <class> oracle.adf.share.security.authorization.MethodPermission </class>
  <name>method_2</name>
  <actions>invoke</actions>
</permission>

<permission>
  <class> oracle.adf.share.security.authorization.MethodPermission </class>
  <name>method_3</name>
  <actions>invoke</actions>
</permission>

<permission>
  <class> oracle.adf.share.security.authorization.MethodPermission </class>
  <name>method_4</name>
  <actions>invoke</actions>
</permission>
...
...
```

As there are potentially many more individual methods for which the `anonymous-role` role must be granted the `invoke` permission, you can greatly simplify the policy definition by replacing the name with a regular expression that represents the set of methods to which `anonymous-role` is granted the `invoke` permission.

For example, the previous listing of permissions could be replaced with a single permission, as shown in [Example 28–3](#).

Example 28–3 Using Regular Expressions to Define Permission for Methods

```
<grant>
  <grantee>
    <principals>
      <principal>
        <class>oracle.security.jps.service.policystore.ApplicationRole</class>
        <name>anonymous-role</name>
      </principal>
    </principals>
  </grantee>
  <permissions>
    <permission>
      <class oracle.adf.share.security.authorization.MethodPermission </class>
      <name>method_*</name>
      <actions>invoke</actions>
    </permission>
  </permissions>
</grant>
```

As the overview editor for the `jazn-data.xml` file does not support the use of regular expressions in the user interface, you must edit the file directly. Do not edit the policy store of the `system-jazn-data.xml` file directly. Instead, add grants using regular expressions to the `jazn-data.xml` file. These grants will then be merged to the policy store when you run or deploy the application.

The use of more complex regular expressions enables you to define business rules in the policy, thus creating a very targeted set of permissions. For example, you can grant the invoke permission on all methods and deny specific methods at the same time by defining an exclusion set in your regular expression. [Example 28–4](#) shows how the invoke permission is granted to the `anonymous-role` role for all methods except those for which the method name starts with `delete`.

Example 28–4 Granting the Invoke Permission to the `anonymous-role` Principal for Specific Methods

```
<grant>
  <grantee>
    <principals>
      <principal>
        <class>oracle.security.jps.service.policystore.ApplicationRole</class>
        <name>anonymous-role</name>
      </principal>
    </principals>
  </grantee>
  <permissions>
    <permission>
      <class>oracle.adf.share.security.authorization.MethodPermission</class>
      <name>[^(&#033;delete)].*</name>
      <actions>invoke</actions>
    </permission>
  </permissions>
</grant>
```

[Table 28–7](#) shows some of the basic regular expression metacharacters that you can use in your policy definitions.

Table 28–7 Description of Metacharacters

Metacharacter	Description
[abc]	a, b, or c (included in list)
[^abc]	Any character except a, b, or c (negation)
[a-zA-Z]	a to z or A to Z, inclusive (range)
[a-d[m-p]]	a to d, or m to p ~=[a-dm-p](union)
[a-z&&[def]]	d, e, or f (intersection)
[a-z&&[^bc]]	a through z, without b and c: [ad-z] (subtraction)
[a-z&&[^m-p]]	a through z, and not m through p
*	Any number of arbitrary characters

28.5 Handling User Authentication in a Fusion Web Application

Oracle ADF Security allows for implicit and explicit authentication:

- In an *implicit authentication* scenario, authentication is triggered automatically if the user is not yet authenticated and they try to access a page that is not granted to the anonymous-role role. After the user successfully logs in, another check will be done to verify whether the authenticated user has view access to the requested page.
- In an *explicit authentication* scenario, a public page has a login link, which, when clicked, triggers an authentication challenge to log in the user. The login link may optionally specify some other target page that should be displayed (assuming the authenticated user has access) after the successful authentication.

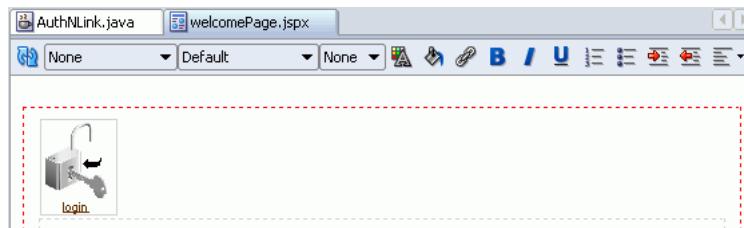
On the first access to a page, if there is no Subject defined, one is created containing the anonymous user principal and the anonymous-role principal.

With this role principal, the user can access any page on which no web.xml security constraint is defined for the anonymous-role principal and on which the view permission has been granted.

For a discussion on how ADF Security lets you grant privileges to the anonymous user, see [Section 28.5.5, "What You May Need to Know About the Anonymous User"](#).

28.5.1 How to Create a Login Component for Your Application

You can create a standard login component that can be added to any page in your application to enable users to authenticate or subsequently log off. This component keeps track of the authenticated state of the user and returns the appropriate login or logout URLs and icons. Furthermore, it keeps track of the name of the current user (anonymous or the name of the logged-in user). Hence, using this login component provide you with a single, consistent object. [Figure 28–31](#) shows a login icon added to the global menu facet of the welcome page in a Fusion web application.

Figure 28–31 Login Icon on the Page

The login component will redirect users back to the current page once they are authenticated.

Note: You may want to alter the component's code to redirect to a welcome page if the current page is not publicly accessible.

To create a login component:

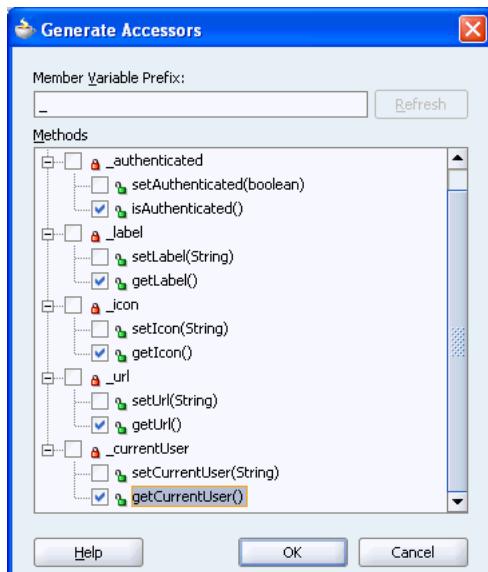
1. In the Applications Navigator, expand the WEB-INF node in the user interface project and double-click the **faces-config.xml** file.
2. In the Structure window, select the **Overview** tab.
3. Right-click **Managed Beans** and choose **Insert managed-bean**.
4. In the Create Managed Bean dialog, specify authNLink for the name, `view.util.AuthNLink` for the class, and set the scope to **session**, as shown in [Figure 28–32](#). Select **Generate Class If It Does Not Exist**, if it is not already selected.

Figure 28–32 Create Managed Bean Dialog

5. Click **OK** and open the `view.util.AuthNLink.java` file under the application sources.
6. Add the following private variables to the managed bean definition, as shown in the following example:

```
public class AuthNLink {
    private boolean _authenticated = false;
    private String _label = null;
    private String _icon = null;
    private String _url = null;
    private String _currentUser = null;
```

7. From the **Source** dropdown list, select **Generate Accessors**.
8. Expand each node and check the getter methods (the ones that start with `is` and `get`), as shown in [Figure 28–33](#).

Figure 28–33 Generate Accessors Dialog

9. Define the methods, as shown in [Example 28–5](#).

Note: Import the ADFContext and FacesContext when prompted, by pressing ALT-Enter with the cursor over the appropriate line.

This example has fixed English labels. To internationalize the code, you can define these strings in a resource bundle. For more information about internationalization, see the "Internationalizing and Localizing Pages" chapter in the *Oracle Fusion Middleware Web User Interface Developer's Guide for Oracle Application Development Framework*.

Example 28–5 Login Component Code

```
// ===== User's Authenticated Status =====
public boolean isAuthenticated() {
    _authenticated =
        ADFContext.getCurrent().getSecurityContext().isAuthenticated();
    return _authenticated;
}

// ===== Link Label =====
public String getLabel()
{
    // toggle link text based on authenticated state of the user.
    if (isAuthenticated())
        { _label = "Click here to log in"; }
    else
        { _label = " Click here to log out"; }
    return _label;
}

// ===== Link Icon =====
public String getIcon()
{
    // toggle icon based on authenticated state of the user.
    if (isAuthenticated())
        _icon = "logout.gif";
    else
```

```

        _icon = "login.gif";
        return (_icon);
    }
// ===== Link URL =====
public String getUrl()
{
    String currentPage = null;
    String urlBaseRef = null;
    String urlBaseRef2 = null;
    FacesContext fctx = FacesContext.getCurrentInstance();
    currentPage = "/faces" + fctx.getViewRoot().getViewId();
    if (isAuthenticated())
        _url = "/adfAuthentication?logout=true&end_url=" + currentPage;
    else
        _url = "/adfAuthentication?success_url=" + currentPage;
    return (_url);
}
// ===== Current User's Name/PrincipalName =====
public String getCurrentUser() {
    _currentUser = ADFContext.getCurrent().getSecurityContext().getUserName();
    return _currentUser;
}

```

In this code, the component uses the `isAuthenticated` method of the Oracle ADF security context to determine whether the user is currently authenticated. The component also modifies the link text, the link text URL, and the associated icon accordingly.

10. Copy your login and logout image files (GIF, JPG, or PNG files) to the `public_html` directory of your project.

Note: The images used should reference the appropriate skin image if your application uses skins. For more information about skins, see the "Customizing the Appearance Using Styles and Skins" chapter in the *Oracle Fusion Middleware Web User Interface Developer's Guide for Oracle Application Development Framework*.

How to Add the Login Component to a Page

To add a login component to a page:

1. In the Application Navigator, double-click the page.
2. From the Component Palette, select **ADF Faces Core**.
3. Select a `menuButtons` component and drag it onto the page.

Note: If you are using an Oracle ADF PanelPage component to lay out the page, you should place the global navigation items such as the login link in the `menuGlobal` facet. This will place the link in a consistent location on the page (by default, this is the top-right corner of the page).

4. Select a `goMenuItem` and drag it onto the `MenuButtons` component.
5. In the Property Inspector, set the `Text`, `Destination`, and `Icon` properties of the `goMenuItem` to the values provided in the following table:

Property	Value
Text	#{authNLink.label}
Destination	#{authNLink.url}
Icon	#{authNLink.icon}

To set these properties, navigate to the authNLink node under JSF Managed Beans in the Binding to Data dialog and set the values provided in the table. You can access the Bind to Data dialog in either of the following ways:

- Right-click **goMenuItem** in the Structure pane and click **Properties**. In the Properties dialog, click the **Bind to Data** icon next to the property.
- In the Property Inspector, click the **Bind to Data** icon in the field next to the property.

[Figure 28–34](#) and [Figure 28–35](#) show the settings for the Text and Destination properties in the Bind to Data dialog.

Figure 28–34 Bind to Data Dialog for the Text Property

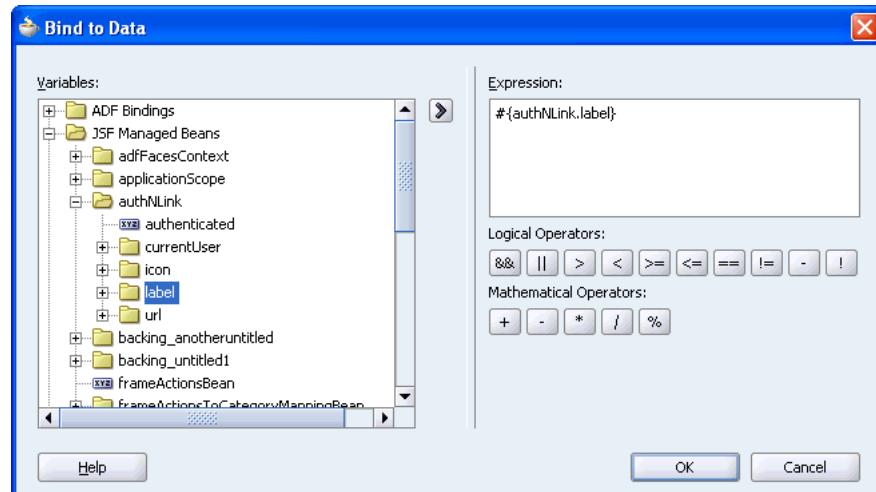
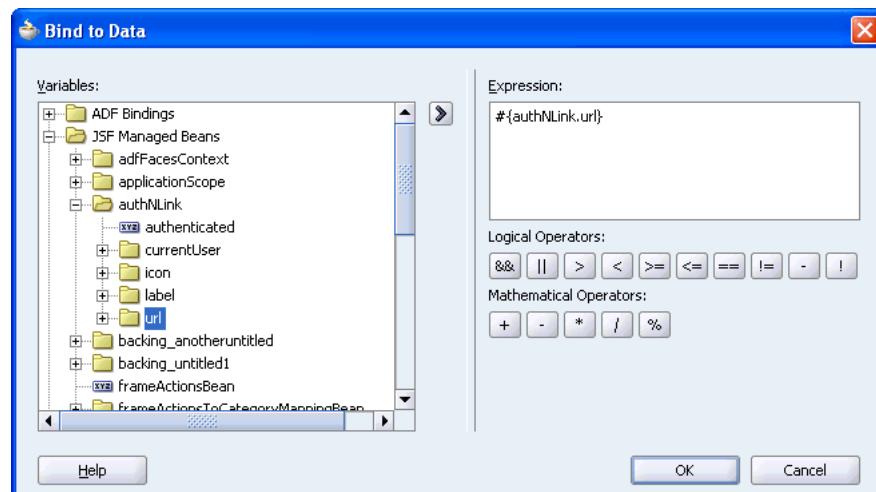
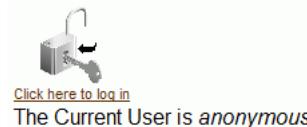


Figure 28–35 Bind to Data Dialog for the Destination Property



6. As the login component keeps track of the current user, you can also display the user's name on your page. To do this, from the Component Palette, select **ADF Faces Core** and drag an **OutputFormatted** component onto the page.
7. Set the value of the **OutputFormatted** to **The Current User is <i>#{authNLink.currentUser}</i>**.
8. Save the page and run it. It will look similar to [Figure 28–36](#).

Figure 28–36 Page with a Log In Link



Note: To enforce security in your application, you must first perform the steps described in [Section 28.2, "Choosing ADF Security Authentication and Authorization"](#) and [Section 28.4, "Defining ADF Security Access Policies"](#). You must, at a minimum, grant view privileges to the *anonymous* role.

9. Click **Log in** and log in as a user with the appropriate credentials. Once logged in, the page will look similar to [Figure 28–37](#).

Figure 28–37 Page with a Log Out Link



28.5.2 How to Add a Login Component to a Web Page

You can add the login component to any page in your application.

To add the login component to a page:

1. In the Application Navigator, double-click the page.
2. From the Component Palette, select **ADF Faces Core**.
3. Select a **menuButtons** component and drag it onto the page.

Note: If you are using an Oracle ADF PanelPage component to lay out the page, you should place global navigation items such as the login link in the `menuGlobal` facet. This will place the link in a consistent location on the page (by default, this is the top-right corner of the page).

4. Select a **goMenuItem** and drag it onto the **MenuButtons** component.
5. In the Property Inspector, set the **Text**, **Destination**, and **Icon** properties of the **goMenuItem** to the values provided in the following table:

Property	Value
Text	# {authNLink.label}
Destination	# {authNLink.url}
Icon	# {authNLink.icon}

To set these properties, navigate to the **authNLink** node under **JSF Managed Beans** in the Bind to Data dialog and set the values provided in the table. You can access the Bind to Data dialog in either of the following ways:

- Right-click **goMenuItem** in the Structure window and click **Properties**. In the Properties dialog, click the **Bind to Data** icon next to the property.
- In the Property Inspector, click the **Bind to Data** icon in the field next to the property.

[Figure 28–34](#) and [Figure 28–35](#) show the settings for the Text and Destination properties in the Bind to Data dialog.

Figure 28–38 Bind to Data Dialog for the Text Property

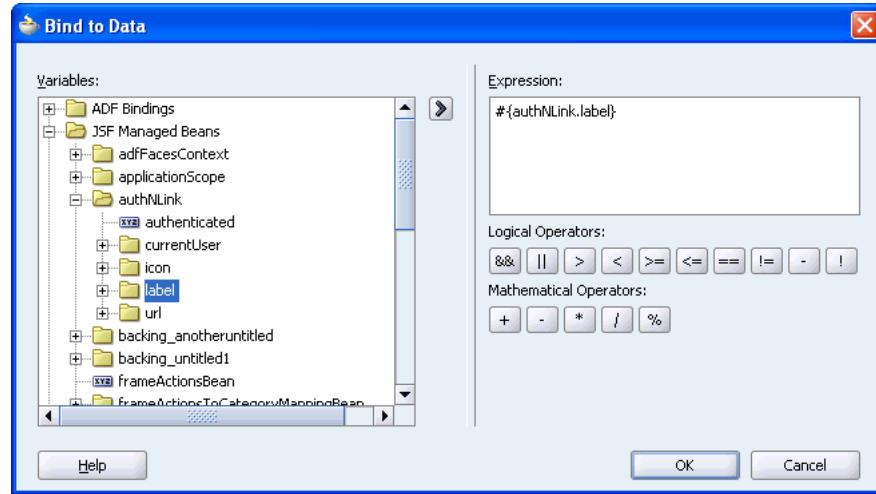
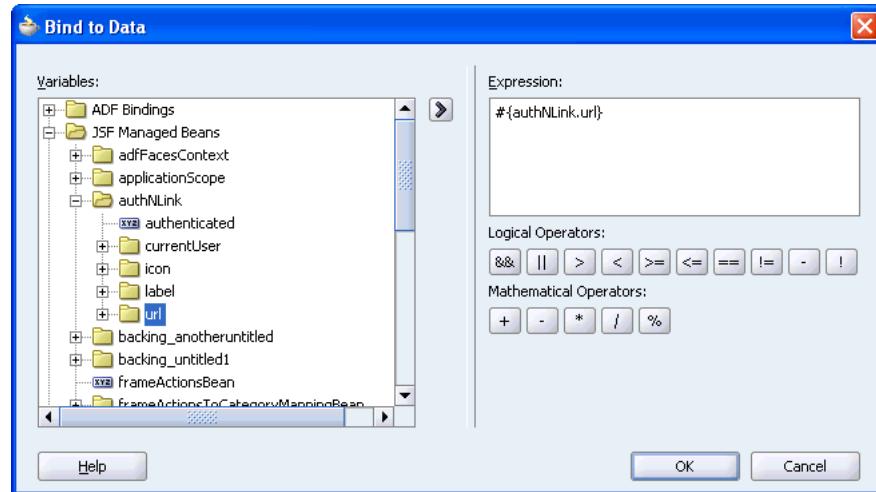
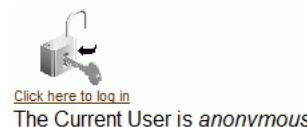


Figure 28–39 Bind to Data Dialog for the Destination Property



6. Because the login component keeps track of the current user, you can also display the user's name on your page. To do this, from the Component Palette, select **ADF Faces Core** and drag an **OutputFormatted** component onto the page.
7. Set the value of the **OutputFormatted** to **The Current User is <i>#{authNLink.currentUser}</i>**.
8. Save the page and run it. It will look similar to [Figure 28–36](#).

Figure 28–40 Page with a Log In Link



Note: To enforce security in your application, you must first perform the steps described in [Section 28.2, "Choosing ADF Security Authentication and Authorization"](#) and [Section 28.4, "Defining ADF Security Access Policies"](#). You must, at a minimum, grant View privileges to the anonymous role.

9. Click **Log in** and log in as a user with the appropriate credentials. Once you are logged in, the page will look similar to [Figure 28–37](#).

Figure 28–41 Page with a Log Out Link



28.5.3 How to Create a Login Page for Your Application

Web applications typically have a notion of public pages and allow for explicit as well as implicit authentication. This means that users can log in to the application by clicking the login link before they navigate to secured content, or they can navigate to a secured page, which will redirect them to the login page for the application. For more information about implicit and explicit authentication, see [Section 28.2.8, "What Happens at Runtime: How Oracle ADF Security Handles Authentication"](#). [Figure 28–42](#) shows a sample login page. The addition of portlets to the login page allows the login page itself to be indistinguishable from the other pages in your Fusion web application.

Note: This section discusses creating an Oracle ADF Faces-based login page that enables you to include customizable components and portlets. However, if adding these components is not a requirement, then a simple JSP or HTML login page can be also used.

Container-based authentication relies on the `j_SecurityCheck` method within the container's security model. Both the Oracle ADF Faces--based login page and the simplified login pages use this method to enforce authentication.

For details about generating a simple login page when running the Configure ADF Security wizard, see [Section 28.2, "Choosing ADF Security Authentication and Authorization"](#).

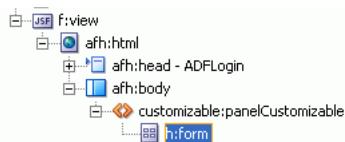
Figure 28–42 Login Page



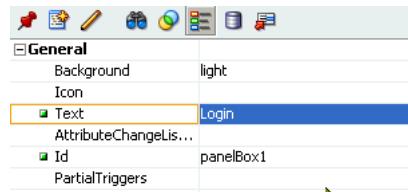
28.5.3.1 Creating an Oracle ADF Faces--Based Login Page

To create the Oracle ADF Faces-based login page:

1. In the Application Navigator, under the user interface project, right-click your application and choose **New**.
2. In the New Gallery dialog, expand the Web Tier node, select **JSF** and in the Items list select **JSF**, then click **OK**.
3. In the Create JSF Page dialog, select **Create as XML Document (*.jspx)**.
4. In the **File Name** field, specify a name for your login page.
5. Expand **Page Implementation** to display the component binding options.
6. Select **Automatically Expose UI Components in a New Managed Bean**.
7. Click **OK**.
8. Save the page.
9. From the Component Palette, select **Customizable Components Core**.
10. Select the **PanelCustomizable** component and drag it onto the Structure window above the **h:form** node.
11. In the Confirm Add Form Element dialog, click **No**. You will be creating a custom HTML form in **PanelBox** instead.
12. In the Property Inspector, set the layout of **Panel Customizable** to **Horizontal**.
13. In the Structure window, drag the **h:form** node onto the **cust:panelCustomizable** node, as shown in [Figure 28–43](#).

Figure 28–43 h:form Node

14. Open the Component Palette, select **ADF Faces Core** and drag a **PanelBox** above the **h:form** tag in the Structure window.
15. Set the **Text** property of the **PanelBox** to **Login**, as shown in [Figure 28–44](#).

Figure 28–44 Text Property of the panelBox

16. Select an **OutputText** component and drag it onto the **PanelBox** component.
17. Save the page.

You can now use a backing bean to inject the appropriate login form into this **PanelBox** area.

28.5.3.2 Creating Login Code for the Backing Bean

After you create the login page as an Oracle ADF Faces page, you cannot just add the login form using form elements from the Component Palette. This would cause the form elements to be serialized and remapped at runtime by the Oracle ADF Faces lifecycle. Instead, you can inject the HTML for the login form at runtime by including it in the backing bean and dynamically showing this at runtime.

To include the HTML code in a backing bean and to reference the HTML login form in the login page:

1. In the Applications Navigator, expand the **Application Resources** node and open the *pageName.java* backing bean.
2. To define the bean **_loginFormBlock**, create a new attribute by adding the following in the declaration section of the *ADFLLogin.java* file:


```
private String _loginFormBlock;
```
3. Add the **get** method for this attribute right before the closing brace **}** in this Java class, as shown in [Example 28–6](#).

Example 28–6 LoginFormBlock Code That Injects the Login Form into the Login Page

```

public String getLoginFormBlock()
{
    String htmlBlock      = null;
    String userNameLabel = "Username";
    String passwordLabel = "Password";
    String buttonLabel   = "Login";
  
```

```

htmlBlock = "\n\n" +
    "<!-- === Login Form Block Generated in Backing Bean ===== -->\n" +
    "<form name=\"LoginForm\" id=\"LoginForm\" \n" +
    "    action=\"j_security_check\" method=\"POST\" >\n" +
    "    <table cellspacing=\"5\" cellpadding=\"0\" border=\"0\" width=\"50%\">\n" +
    "        <tr>\n" +
    "            <td nowrap>" + userNameLabel + "</td>\n" +
    "            <td nowrap><input type=\"text\" name=\"j_username\" /></td>\n" +
    "        </tr>\n" +
    "        <tr>\n" +
    "            <td nowrap>" + passwordLabel + "</td>\n" +
    "            <td nowrap><input type=\"password\" name=\"j_password\" /></td> \n" +
    "        </tr>\n" +
    "        <tr><input type=\"submit\" value=\"\" + buttonLabel + "\"/></td> \n" +
    "    </tr>\n" +
    "    </table>\n" +
    "</form>\n" +
    "<!-- ===== -->\n\n" ;
_loginFormBlock = htmlBlock;
return (_loginFormBlock);
}

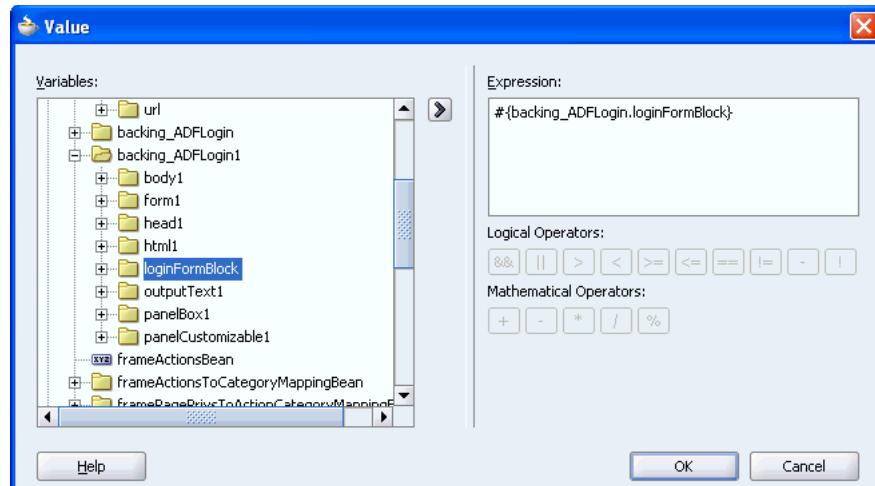
```

This code returns the entire login block as a simple output string (simplifying the need for `<verbatim>` tags).

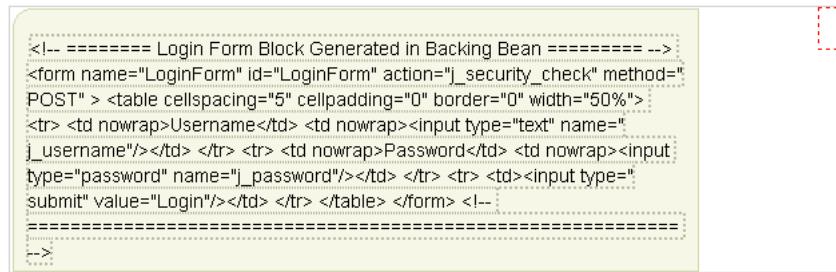
4. Save the Java file.
5. In the Property Inspector, set the value of the previously created `outputText` object to the following value of the `loginFormBlock` attribute, as shown in [Figure 28–45](#):

```
#{{backing_ADFLogin.loginFormBlock}}
```

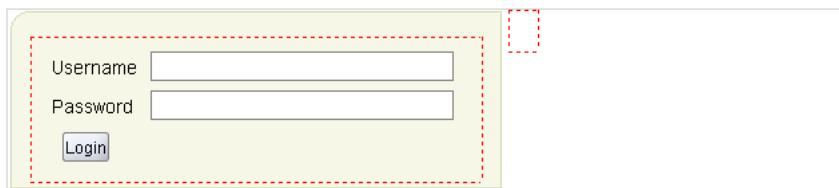
Figure 28–45 *loginFormBlock* Attribute



This will result in the full sting appearing as "Text" within the page, as shown in [Figure 28–46](#).

Figure 28–46 Login Form Block Generated in the Backing Bean

- In the Property Inspector, set the **Escape** property of the **outputText** object to **False** to render the string as static HTML as shown in [Figure 28–47](#).

Figure 28–47 HTML Login Form

- Save the file.

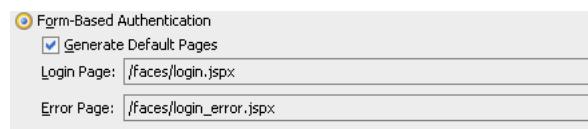
28.5.3.3 Configuring the web.xml File for an Oracle ADF Faces-Based Login Page

Because the login page is called directly from the container, it is not part of the Oracle ADF Faces navigation process. As such, you must force a call to the Oracle ADF Faces servlet when calling the login page.

You can accomplish this in the Authentication Type page of the Configure ADF Security wizard when you configure ADF Security or in the `web.xml` file directly. If you have already run the Configure ADF Security wizard, you can use the following procedure to confirm that the `web.xml` file has been updated as described.

To reference an login page as part of the ADF Faces lifecycle:

- In the Application Navigator, expand the **WEB-INF** node, right-click **web.xml** and choose **Properties**.
- In the Web Application Deployment Descriptor dialog, select **Login Configuration**.
- In the Login Configuration page, set the login page to include a reference to the Oracle ADF Faces servlet such that the login page can be part of the Oracle ADF Faces lifecycle `/faces/ADFlogin.jspx`, as shown in [Figure 28–48](#).

Figure 28–48 Adding a Reference to the Faces Servlet in the Login Configuration

- Click **OK**.

28.5.3.4 Ensuring That the Login Page Is Public

Because the application is secured by Oracle ADF Security, all web pages defined within bounded task flows are secured or any web page defined by an ADF page definition. Since all users must be allowed to log on, the login page should remain publicly accessible. No further steps are required to ensure that the container will always redirect to the defined authentication point before allowing access to the page (which in this case is the authentication page).

Use only ASCII characters to name applications and projects when you enable ADF Security. Login will fail when either the application name or the project name contains multibyte characters.

28.5.4 How to Create a Public Welcome Page for Your Application

Because web applications are generally secured, there is always a need for a starting point or home page for unauthenticated users. To create this public welcome page, you create an Oracle ADF Faces page to act as the entry point for the application, which contains links to other pages within the application. However, only links to public pages should be rendered to unauthenticated users and, conversely, links to secured pages should be rendered only after the user has logged in and has the appropriate privileges to view the target page.

Note: When you run the Configure ADF Security wizard, you can optionally allow the wizard to generate a default welcome page for your application. For details about running the wizard, see [Section 28.2, "Choosing ADF Security Authentication and Authorization"](#).

28.5.4.1 Ensuring That the Welcome Page Is Public

After you have created a regular Oracle ADF Faces page, the page will, by default, be public and accessible by unauthenticated users. If, however, you have associated the welcome page with an ADF resource, for example, by dropping databound ADF Faces components into the welcome page using the Data Controls Panel, then ADF Security will secure the page by default. You can make any ADF resource publicly accessible using the overview editor for ADF security policies by granting a View privilege on the resource to the provided anonymous-role. For details about the anonymous-role see, [Section 28.5.5, "What You May Need to Know About the Anonymous User"](#).

28.5.4.2 Adding Login and Logout Links

You can add login and logout links to your public welcome page so that users can explicitly log in and out while they are in the application. While Java EE container-managed security supports the concept of authentication when accessing a secured resource, there is no standard way to log out and stay within a secured application. However, it is a common practice in web applications to allow the user to stay on the same page if that page is public or to return the user to the welcome page if that page is secured. While adding the login and logout links to each page would let the user end their login session anywhere within the application (and return to the welcome page), having these links on the welcome page enables users to explicitly authenticate on entering the application.

To add the login and logout links, you must add a login component to your application and then add the login and logout links to your page, as described in [Section 28.5.1, "How to Create a Login Component for Your Application"](#).

28.5.4.3 Hiding Links to Secured Pages

Since an anonymous user should not have access to any secured pages, any navigation component on the welcome page that points to a secured page should be hidden from view based on the following two criteria:

- Is the user authenticated with a known user identity?
- Does the specified user identity have permission to view the target?

If either of these criteria has not been met, the `rendered` attribute of any navigation component on a public page that points to a secured resource must have its `rendered` property set to `false`, thus hiding it from the anonymous user. To enforce these rules within your welcome page, see section [Section 28.6, "Performing Authorization Checks in a Fusion Web Application"](#).

28.5.5 What You May Need to Know About the Anonymous User

It is a common requirement that some web pages should be available to all users regardless of their specific access privileges. For example, the home page should be seen by all visitors to the site, while a corporate site should be available only to those who have identified themselves through authentication.

In both cases, the page may be considered public, because the ability to view the page is not defined by the users' specific permissions. Rather, the difference is whether the user is anonymous or a known identity.

This use of public is different from traditional Java EE security, which does not let you distinguish between completely unsecured content (security has not been enabled) and public content.

In the Oracle ADF Security model, you explicitly differentiate between the absence of security and public access to content by granting access privileges to the `anonymous-role` principal. The `anonymous-role` is a role that encompasses both known and anonymous users thus, permission granted to `anonymous-role` allows access to a resource by unauthenticated users, for example, guest users. To provide access to authenticated users only, the policy must be defined for the `authenticated-role` principal. For details about granting to these built-in roles, see [Section 28.3.3, "How to Configure the Identity Store with Test Users in JDeveloper"](#).

28.6 Performing Authorization Checks in a Fusion Web Application

While the existence of a policy will prevent unauthorized users from accessing a secured resource, their attempt to access the resource would result in a security exception. Good security practice dictates that users should not be aware of resources and capabilities to which they do not have access.

For example, if a user does not have permission to view an administrative page, then all navigation components that point to that page should be dynamically removed for that user. Similarly, if a user does not have permission to begin a new task flow, a button that the page displays to invoke that task flow should not be visible to the user.

While the application must first evaluate the policy to determine whether the user has the appropriate permission required by the target resource, ultimately the ability to attempt access to a secured resource or function (such as a delete button) is controlled by the UI component's `Rendered` property.

By default, the `Rendered` property is set to `true`. By dynamically changing this value based on the permission, the UI component can be shown or hidden. For example, if the user has the appropriate permission, the `Rendered` property should be set to

true so that the UI component is shown. If they do not have permission, the property should be set to false and the UI component hidden from view.

Note: The ability to evaluate a policy is limited to the current request. For this reason, it is important to understand where the policy evaluation occurs, because evaluating the policy at anything other than the request scope can lead to unexpected results.

You can use Expression Language (EL) to evaluate the policy directly in the UI, while the use of Java enables you to evaluate the policy from within a managed bean. ADF Security implements several convenience methods for use in EL expressions to access ADF resources in the security context. For example, you can use the EL expression convenience methods to determine whether the user is allowed to access a particular task flow.

28.6.1 How to Evaluate Policies Using Expression Language (EL)

The use of EL within a UI element allows for properties to be defined dynamically, resulting in modification of the UI component at runtime. In the case of securing resources, the UI property of interest is the Rendered property, which allows you to show and hide components based on available permissions.

To evaluate a policy using EL, you must use the ADF Security methods in the security context (# {securityContext...}). These methods let you access information in the ADF security context for a particular user or ADF resource.

Table 28–8 shows the EL expression that is required to determine whether a user has the associated permission. If the user has the appropriate permission, the EL expression evaluates to true; otherwise, it returns false.

Table 28–8 EL Expression to Determine View Permissions on ADF Resources

Expression	Expression action
# {securityContext.taskflowViewable[MyTaskFlow] } For example: # {securityContext.taskflowViewable[/WEB-INF/audit-expense-report.xml#audit-expense-report] }	Where MyTaskFlow is the WEB-INF node-qualified name of the task flow being accessed. Returns true if the user has access rights. Returns false if the user does not have sufficient access rights.
# {securityContext.regionViewable[MyPagePageDef] }	Where MyPagePageDef is the qualified name of the page definition file associated with the web page being accessed. Returns true if the user has access rights. Returns false if the user does not have sufficient access rights.

Note: In the case of page permission, the value of the page definition can be specified dynamically by using late-binding EL within a managed bean, as described in [Section 28.2.3, "What You May Need to Know About the valid-users Role"](#).

Table 28–9 shows the EL expression that lets you get general information from the ADF security context not related to a particular ADF resource. For example, you can access the current user name when you want to display the user's name in the user interface. You can also check whether the current user is a member of certain roles or

granted certain privileges. Your application may use this result to dynamically hide or show menus.

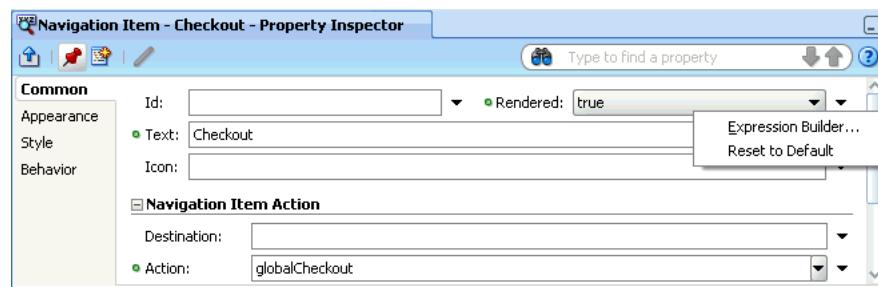
Table 28–9 EL Expression to Determine User Information in ADF Security Context

Expression	Expression action
<code>#{securityContext.userName}</code>	Returns the user name of the authenticated user.
<code>#{securityContext.userInRole['roleList']}</code>	Where <code>roleList</code> is a comma-separated list of role names. Returns <code>true</code> if the user is in at least one of the roles. Returns <code>false</code> if the user is in none of the roles, or if the user is not currently authenticated.
<code>#{securityContext.userInAllRoles['roleList']}</code>	Where <code>roleList</code> is a comma-separated list of role names. Returns <code>true</code> if the user is in all of the roles. Returns <code>false</code> if the user is not in all of the roles, or if the user is not currently authenticated.
<code>#{securityContext.userGrantedPermission['permission']}</code>	Where <code>permission</code> is a string containing a semicolon-separated concatenation of <code>permissionClass=<class>;target=<artifact_name>;action=<action></code> . Returns <code>true</code> if the user has access rights. Returns <code>false</code> if the user does not have sufficient access rights.
	Note that the convenience methods <code>taskflowViewable</code> and <code>regionViewable</code> shown in Table 28–8 provide the same functionality.

To associate the rendering of a navigation component to a user's granted permissions on a target task flow or page definition:

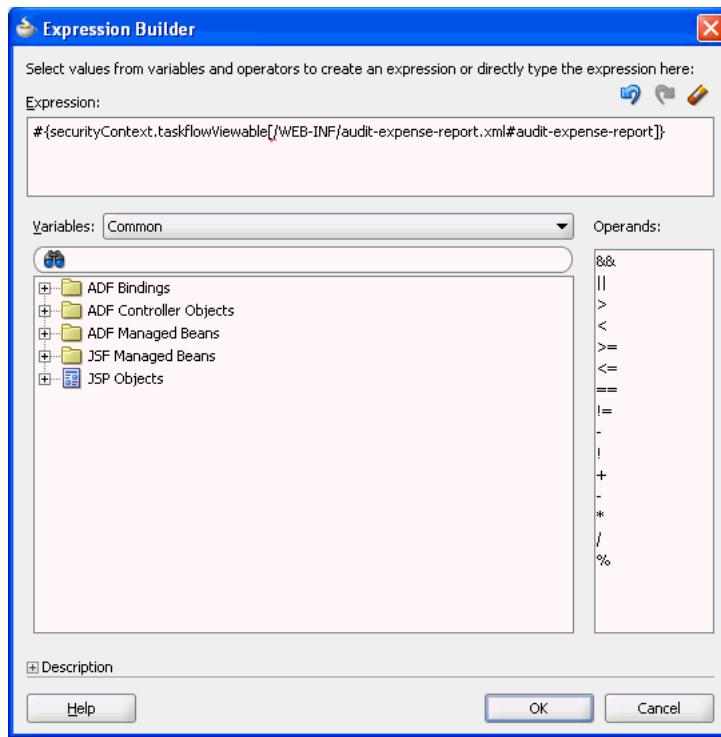
1. In the Application Navigator, double-click the page.
2. Select the component that is used to navigate to the secured page.
3. In the Property Inspector, select **Expression Builder** from the dropdown menu displayed to the right of the **Rendered** property, as shown in [Figure 28–49](#).

Figure 28–49 Binding the Rendered Property to Data



4. Enter the appropriate EL expression for the ADF resource that the user will attempt to access.

For example, to limit access to a task flow, you would enter an expression `#{securityContext.taskflowViewable['target']}` like the one shown in [Figure 28–50](#). In this example, audit-expense-report is the secured target task flow.

Figure 28–50 Defining EL in the Bind to Data Dialog

5. Click **OK**.

When you run the application, the component will be rendered or hidden based on the user's ability to view the target page.

28.6.2 What You May Need to Know About Delayed Evaluation of EL

The ability to evaluate a security permission is scoped to the request. If you want to evaluate permissions to access a target page from a managed bean that is scoped to a higher level than Request (for example, a global menu that is backed by a session-scoped managed bean), you must implement delayed EL evaluation (late-binding). By passing in the target page as a managed property of the bean, you ensure that the EL expression is evaluated only after the required binding information is available to the session-scoped bean. Because EL is evaluated immediately when the page is executed, placing the EL expression directly in the properties of a UI component, backed by a session-scoped bean, would result in an out-of-scope error.

[Example 28–7](#) shows a property (`authorized`) of a session-scoped bean that returns true or false based on a user's ability to view a named target page. In this case, the `_targetPageDef` variable is a managed property containing the name of the target page. Within the UI, the EL expression would reference the `authorized` property, rather than `securityContext.regionViewable`.

Example 28–7 Delayed EL Evaluation in a Session-Scoped Managed Bean

```
public boolean isAuthorized()
{
    if (_targetPageDef != null) {
        FacesContext ctx = FacesContext.getCurrentInstance();
        ValueBinding vb = ctx.getApplication().createValueBinding(
            "#{securityContext.regionViewable['" + _targetPageDef + "']}");
        if (vb != null) {
```

```

        Object authResult = vb.getValue(ctx);
        return (Boolean) authResult;
    }
    else {
        ctx.addMessage(null, new FacesMessage (
            FacesMessage.SEVERITY_WARN, "Access Permission not defined! " , null));
        return(true);
    }
}

```

28.6.3 How to Evaluate Policies Using Java

To evaluate the security policies from within Java, you can use the `hasPermission` method of the Oracle ADF Security context. This method takes a permission object (defined by the resource and action combination) and returns `true` if the user has the corresponding permission.

In [Example 28–8](#), a convenience function is defined to enable you to pass in the name of the page and the desired action, returning `true` or `false` based on the user's permissions. Because this convenience function is checking page permissions, the `RegionPermission` class is used to define the permission object that is passed to the `hasPermission` method.

Example 28–8 Using the hasPermission Method to Evaluate Access Policies

```

private boolean TestPermission (String PageName, String Action) {
    Permission p = new RegionPermission("view.pageDefs." + PageName + "PageDef",
                                         Action);
    if (p != null) {
        return ADFContext.getCurrent().getSecurityContext().hasPermission(p);
    }
    else {
        return (true);
    }
}

```

As it is possible to determine the user's permission for a target page from within a backing bean, you can now use this convenience method to dynamically alter the result of a Faces navigation action. In [Example 28–9](#), you can see that a single command button can point to different target pages depending on the user's permission. By checking the View permission from the most secured page (the manager page) to the least secured page (the public welcome page), the command button will apply the appropriate action to direct the user to the page that corresponds to their permission level. The backing bean that returns the appropriate action is using the convenience method defined in [Example 28–8](#).

Example 28–9 Altering a Page Navigation Result Based on a Permission Check

```

//CommandButton Definition
<af:commandButton text="Goto Your Group Home page"
                  binding="#{backing_content.commandButton1}"
                  id="commandButton1"

                  action="#{backing_content.getSecureNavigationAction}" />

//Backing Bean Code
public String getSecureNavigationAction() {
    String ActionName;
    if (TestPermission("ManagerPage", "view"))
        ActionName = "goToManagerPage";

```

```
        else if (TestPermission("EmployeePage", "view"))
            ActionName = "goToEmployeePage";
        else
            ActionName = "goToWelcomePage";
        return (ActionName);
    }
```

28.7 Getting Other Information from the Oracle ADF Security Context

The implementation of security in a Fusion web application is by definition an implementation of the security infrastructure of the Oracle ADF framework. As such, the security context of the framework allows access to information that will be required as you define the policies and the overall security for your application.

28.7.1 How to Determine Whether Security Is Enabled

Because the enforcement of Oracle ADF Security can be turned on and off at the container level independent of the application, you should determine whether Oracle ADF Security is enabled prior to making permission checks. This can be achieved by evaluating the `isAuthorizationEnabled()` method of the Oracle ADF Security context, as shown in [Example 28–10](#).

Example 28–10 Using the `isAuthorizationEnabled()` Method of the Oracle ADF Security Context

```
if (ADFContext.getCurrent().getSecurityContext().isAuthorizationEnabled()){
    //Permission checks are performed here.
}
```

28.7.2 How to Determine Whether the User Is Authenticated

As the user principal in a Fusion web application is never `null` (that is, it is either anonymous for unauthenticated users or the actual user name for authenticated users), it is not possible to simply check whether the user Principal is `null`, to determine if the user has logged on or not. As such, you must use a method to take into account that a user Principal of anonymous indicates that the user has not authenticated. This can be achieved by evaluating the `isAuthenticated()` method of the Oracle ADF security context, as shown in [Example 28–11](#).

Example 28–11 Using the `isAuthenticated()` Method of the Oracle ADF Security Context

```
// ===== User's Authenticated Status =====
private boolean _authenticated;
public boolean isAuthenticated() {
    _authenticated = ADFContext.getCurrent().getSecurityContext().isAuthenticated();
    return _authenticated;
}
```

28.7.3 How to Determine the Current User Name

Fusion web applications support the concept of public pages that, while secured, are available to all users. Furthermore, components on the web pages, such as portlets, require knowledge of the current user identity. As such, the user name in a Fusion web application will never be `null`. If an unauthenticated user accesses the page, the user name `anonymous` will be passed to page components.

You can determine the current user's name by evaluating the `getUserName()` method of the Oracle ADF security context, as shown in [Example 28–12](#). This method

returns the string anonymous for all unauthenticated users and the actual authenticated user's name for authenticated users.

Example 28–12 Using the `getUserName()` Method of the Oracle ADF Security Context

```
// ===== Current User's Name/PrincipalName =====
public String getCurrentUser() {
    _currentUser = ADFContext.getCurrent().getSecurityContext().getUserName();
    return _currentUser;
}
```

Because the traditional method for determining a user name in a Faces-based application

(`FacesContext.getCurrentInstance().getExternalContext().getRemoteUser()`) returns null for unauthenticated users, you need to use additional logic to handle the public user case if you use that method.

28.7.4 How to Determine Membership of a Java EE Security Role

Although Fusion web application security is centered around JAAS policies, you will likely still need to use Java EE security roles to secure components within an application page based on role membership. As Fusion web applications are JavaServer Faces-based applications, you can use the `isUserInRole(roleName)` method of the Faces external context, as shown in [Example 28–13](#), to determine whether a user is in a specified role.

In this example, a convenience method (`checkIsUserInRole`) is defined. The use of this method within a managed bean enables you to expose membership of a named role as an attribute, which can then be used in EL.

Example 28–13 Using the `isUserInRole(roleName)` Method of the Faces Context

```
public boolean checkIsUserInRole(String roleName) {
    return
        (FacesContext.getCurrentInstance().getExternalContext().isUserInRole(roleName));
}

public boolean isTechnician() {
    return (checkIsUserInRole("technicians"));
}
```

28.8 Testing ADF Security with Integrated WLS in JDeveloper

Oracle JDeveloper's Integrated WLS enables you to run the application directly. However, at this time, JDeveloper does not support application-level security in Integrated WLS. When you run the application using Integrated WLS, JDeveloper migrates the `jazn-data.xml` file to the domain-level `system-jazn-data.xml` policy store. In the migration process, JDeveloper maps the Oracle Platform Security (JPS) application role member classes to the WebLogic Server (WLS) member classes and migrates the users to WLS identity store users and roles to WLS identity store groups. In WebLogic Server, `users` is an implicit group equivalent to JPS `authenticated-role`.

Example 28–14 Application Role Fragment in `system-jazn-data.xml` File

```
<app-roles>
    <app-role>
        <name>fod-user</name>
```

```

<guid>FFFF394F696E786F4134485764511002</guid>
<display-name/>
<description/>
<class>oracle.security.jps.service.policystore.ApplicationRole</class>
<members>
    <member>
        <name>fod-user</name>
        <class>weblogic.security.principal.WLSGroupImpl</class>
    </member>
</members>
</app-role>
</app-roles>

```

When you run the application, Integrated WLS executes the `JpsFilter` definition in the `web.xml` file to set up the JPS policy provider. The filter defines settings that indicate that your servlet has special privileges and the `doasprivileged` setting specifies the privileges that are allowed. [Example 28–15](#) shows the filter definition that the Configure ADF Security wizard adds to the `web.xml` file. It is important that the `JpsFilter` is the first filter in the `web.xml` file.

Example 28–15 JpsFilter Definition for ADF Authorization

```

<filter>
    <filter-name>JpsFilter</filter-name>
    <filter-class>oracle.security.jps.ee.http.JpsFilter</filter-class>
    <init-param>
        <param-name>enable.anonymous</param-name>
        <param-value>true</param-value>
    </init-param>
    <init-param>
        <param-name>remove.anonymous.role</param-name>
        <param-value>false</param-value>
    </init-param>
    <init-param>
        <param-name>addAllRoles</param-name>
        <param-value>true</param-value>
    </init-param>
    <init-param>
        <param-name>jaas.mode</param-name>
        <param-value>doasprivileged</param-value>
    </init-param>
</filter>
<filter-mapping>
    <filter-name>JpsFilter</filter-name>
    <url-pattern>*.jspx</url-pattern>
    <dispatcher>FORWARD</dispatcher>
    <dispatcher>REQUEST</dispatcher>
</filter-mapping>
<filter-mapping>
    <filter-name>JpsFilter</filter-name>
    <url-pattern>*.jsp</url-pattern>
    <dispatcher>FORWARD</dispatcher>
    <dispatcher>REQUEST</dispatcher>
</filter-mapping>

```

28.9 Preparing for Deployment to a Production Environment

Deploying a Fusion web application to a production application server involves migrating application-level policy data to a domain-level policy store. Your application stores the application-level policies in the `jazn-data.xml` file-based repository. Once migrated to the domain-level policy store, the policy data will be merged into the `system-jazn-data.xml` file.

You perform this task outside of JDeveloper, using the supplied `migrateSecurityStore` command. For details about using this command to migrate your application's policy data to the domain-level policy store, see "Migrating the Security Repository to a Production Environment" in the "Developing Secure Applications" section of the JDeveloper online help.

29

Testing and Debugging ADF Components

This chapter describes the process of debugging your user interface project. It describes several JDeveloper tools that can be used to help debug an application. It contains debugging procedures with breakpoints using the ADF Declarative Debugger. Finally, it explains how to write and run regression tests for your ADF Business Components-based business services.

This chapter includes the following sections:

- [Section 29.1, "Introduction to Oracle ADF Debugging"](#)
- [Section 29.2, "Correcting Simple Oracle ADF Compilation Errors"](#)
- [Section 29.3, "Correcting Simple Oracle ADF Runtime Errors"](#)
- [Section 29.4, "Using the ADF Logger and Business Component Browser"](#)
- [Section 29.5, "Using the ADF Declarative Debugger"](#)
- [Section 29.6, "Setting ADF Declarative Breakpoints"](#)
- [Section 29.7, "Setting Java Code Breakpoints"](#)
- [Section 29.8, "Regression Testing with JUnit"](#)

29.1 Introduction to Oracle ADF Debugging

Like any debugging task, debugging the web application's interaction with Oracle ADF is a process of isolating specific contributing factors. However, in the case of web applications, generally this process does not involve compiling Java source code. Your web pages contain no Java source code, as such, to compile. In fact, you may not realize that a problem exists until you run and attempt to use the application. For example, these failures are only visible at runtime:

- A page not found servlet error
- The page is found, but the components display without data
- The page fails to display data after executing a method call or built-in operation (like Next or Previous)
- The page displays, but a method call or built-in operation fails to execute at all
- The page displays, but unexpected validation errors occur

The failure to display data or to execute a method call arises from the interaction between the web page's components and the Oracle ADF Model layer. When a runtime failure is observed during ADF lifecycle processing, the sequence of preparing the model, updating the values, invoking the actions, and, finally, rendering the data failed to complete.

Fortunately, most failures in the web application's interaction with Oracle ADF result from simple and easy-to-fix errors in the declarative information that the application defines or in the EL expressions that access the runtime objects of the page's Oracle ADF binding container.

In your databound Fusion web application, you should examine the declarative information and EL expressions as likely contributing factors when runtime failures are observed. Read the following sections to understand editing the declarative files:

- [Section 29.2, "Correcting Simple Oracle ADF Compilation Errors"](#)
- [Section 29.3, "Correcting Simple Oracle ADF Runtime Errors"](#)

One of the most useful diagnostic tools is the ADF Logger. You use this logging mechanism in JDeveloper to capture runtime traces messages. With ADF logging enabled, JDeveloper displays the application trace in the Message Log window. The trace includes runtime messages that may help you to quickly identify the origin of an application error. Read [Section 29.4, "Using the ADF Logger and Business Component Browser"](#) to configure the ADF Logger to display detailed trace messages.

Supported Oracle ADF customers can request Oracle ADF source code from Oracle Worldwide Support. This can make debugging Oracle ADF Business Components framework code a lot easier. Read [Section 29.5.1, "Using ADF Source Code with the Debugger"](#) to understand how to configure JDeveloper to use the Oracle ADF source code.

If the error cannot be easily identified, you can utilize the ADF Declarative Debugger in JDeveloper to set breakpoints. When a breakpoint is reached, the execution of the application is paused and you can examine the data that the Oracle ADF binding container has to work with, and compare it to what you expect the data to be.

Depending on the types of breakpoints, you may be able to use the Step functions to move from one breakpoint to another. For more information about the debugger, read [Section 29.5, "Using the ADF Declarative Debugger"](#).

JDeveloper provides integration with JUnit for your Business Components-based Fusion web application through a wizard that generates regression test cases. Read [Section 29.8, "Regression Testing with JUnit"](#) to understand how to write test suites for your application.

29.2 Correcting Simple Oracle ADF Compilation Errors

When you create web pages and work with the ADF data controls to create the ADF binding definitions in JDeveloper, the Oracle ADF declarative files you edit must conform to the XML schema defined by Oracle ADF. When an XML syntax error occurs, the JDeveloper XML compiler immediately displays the error in the Structure window.

Although there is some syntax checking during design time, the JDeveloper compiler is currently limited by an inability to resolve EL expressions. EL expressions in your web pages interact directly with various runtime objects in the web environment, including the web page's Oracle ADF binding container. At present, errors in EL expressions can be observed only at runtime. Thus, the presence of a single typing error in an object-access expression will not be detected by the compiler, but will manifest at runtime as a failure to interact with the binding container and a failure to display data in the page. For information about debugging runtime errors, see [Section 29.3, "Correcting Simple Oracle ADF Runtime Errors"](#).

Tip: The JDeveloper Expression Builder is a dialog that helps you build EL expressions by providing lists of objects, managed beans, and properties. It is particularly useful when creating or editing ADF databound EL expressions because it provides a hierarchical list of ADF binding objects and their valid properties from which you can select. You should use the Expression Builder to avoid introducing typing errors. For details, see [Section 11.7, "Creating ADF Data Binding EL Expressions"](#).

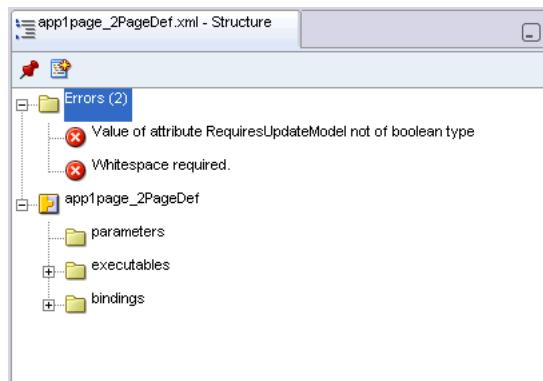
[Example 29–1](#) illustrates two simple compilation errors contained in a page definition file: `tru` instead of `true` and `id="CountryCodesView1Iterator" /` instead of `id="CountryCodesView1Iterator" />` (that is, the ID is missing a closing angle bracket).

Example 29–1 Sample Page Definition File with Two Errors

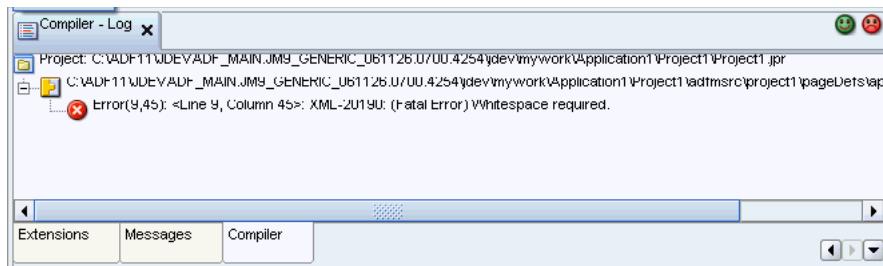
```
<?xml version="1.0" encoding="UTF-8" ?>
<pageDefinition xmlns="http://xmlns.oracle.com/adfm/uimodel"
    version="11.1.1.42.54" id="app1page_2PageDef"
    Package="project1.pageDefs">
<parameters/>
<executables>
    <iterator Binds="CountryCodesView1" RangeSize="10"
        DataControl=" AppModuleDataControl "
        id="CountryCodesView1Iterator" /
    </executables>
    <bindings>
        <action id="Find" RequiresUpdateModel="tru" Action="3"
            IterBinding="CountryCodesView1Iterator" />
    </bindings>
</pageDefinition>
```

During compilation, the Structure window displays the XML errors in the page, as shown in [Figure 29–1](#).

Figure 29–1 Structure Window Displays XML Error



The logger window also displays the compilation errors in the page, as shown in [Figure 29–2](#).

Figure 29–2 Compiler Window Displays XML Compile Error**To view and correct schema validation errors:**

1. From the main menu, choose **View > Structure** to open the Structure window or **View > Log** to open the Log Window.
2. In either window, double-click the error message to open the file in the XML editor.
3. In the XML editor, locate the highlighted lines.
The highlighted lines will be lines with errors.
4. Correct any errors.
After an error has been corrected, the corresponding error message will be automatically removed from the Structure window.
5. Optionally, you can recompile the project by choosing **Run > Make** and checking to see whether the compiler still produces the error message.

29.3 Correcting Simple Oracle ADF Runtime Errors

Failures of the Oracle ADF Model layer cannot be detected by the JDeveloper compiler, in part because the page's data-display and method-execution behavior relies on the declarative Oracle ADF page definition files. The Oracle ADF Model layer utilizes those declarative files at runtime to create the objects of the Oracle ADF binding container.

To go beyond simple schema validation, you will want to routinely run and test your web pages to ensure that none of the following conditions exists:

- The project dependency between the data model project and the user interface project is disabled.

By default, the dependency between projects is enabled whenever you create a web page that accesses a data control in the data model project. However, if the dependency is disabled and remains disabled when you attempt to run the application, an internal servlet error will be generated at runtime:

```
oracle.jbo.NoDefException: JBO-25002: Definition
model.DataControls.dcx of type null not found
```

To correct the error, double-click the user interface project, and select the **Dependencies** node in the dialog. Make sure that the **ModelProjectName.jpr** option appears selected in the panel.

- Page definition files have been renamed, but the `DataBindings.cpx` file still references the original page definition file names.

While JDeveloper does not permit these files to be renamed within the IDE, if a page definition file is renamed outside of JDeveloper and the references in the

DataBindings.cpx file are not also updated, an internal servlet error will be generated at runtime:

```
oracle.jbo.NoDefException: JBO-25002: Definition
oracle.<path>.pageDefs.<pagedefinitionName> of type Form
Binding Definition not found
```

To correct the error, open the DataBindings.cpx file and use the source editor to edit the page definition file names that appear in the <pageMap> and <pageDefinitionUsages> elements.

- The web page file (.jsp or .jspx) has been renamed, but the DataBindings.cpx file still references the original file name of the same web page.

The page controller uses the page's URL to determine the correct page definition to use to create the ADF binding container for the web page. If the page's name from the URL does not match the <pageMap> element of the DataBindings.cpx file, an internal servlet error will be generated at runtime:

```
javax.faces.el.PropertyNotFoundException: Error testing
property <propertyname>
```

To correct the error, open the DataBindings.cpx file and use the source editor to edit the web page file names that appear in the <pageMap> element.

- Bindings have been renamed in the web page EL expressions, but the page definition file still references the original binding object names.

The web page may fail to display information that you expect to see. To correct the error, compare the binding names in the page definition file and the EL expression responsible for displaying the missing part of the page. Most likely the mismatch will occur on a value binding, with the consequence that the component will appear but without data. Should the mismatch occur on an iterator binding name, the error may be more subtle and may require deep debugging to isolate the source of the mismatch.

- Bindings in the page definition file have been renamed or deleted, and the EL expressions still reference the original binding object names.

Because the default error-handling mechanism will catch some runtime errors from the ADF binding container, this type of error can be very easy to find. For example, if an iterator binding named findUsersByNameIter was renamed in the page definition file, yet the page still refers to the original name, this error will display in the web page:

```
JBO-25005: Object name <iterator> for type Iterator Binding
Definition is invalid
```

To correct the error, right-click the name in the web page and choose **Go to Page Definition** to locate the correct binding name to use in the EL expression.

- EL expressions were written manually instead of using the expression picker dialog and invalid object names or property names were introduced.

This error may not be easy to find. Depending on which EL expression contains the error, you may or may not see a servlet error message. For example, if the error occurs in a binding property with no runtime consequence, such as displaying a label name, the page will function normally but the label will not be displayed. However, if the error occurs in a binding that executes a method, an internal servlet error javax.faces.el.MethodNotFoundException: *methodname* will display. Or, in the case of an incorrectly typed property name on the method

expression, the servlet error
javax.faces.el.PropertyNotFoundException: *propertyname* will display.

If this list of typical errors does not help you to find and fix a runtime error, you can initiate debugging within JDeveloper to find the contributing factor. For an ADF application, start setting ADF declarative breakpoints to find the problem. Using the ADF Declarative Debugger to set ADF declarative breakpoints is described in [Section 29.5, "Using the ADF Declarative Debugger"](#) and [Section 29.6, "Setting ADF Declarative Breakpoints"](#). This process involves pausing the execution of the application as it proceeds through the application and examining data. You can also use the ADF Declarative Debugger to set Java code breakpoints, as described in [Section 29.7, "Setting Java Code Breakpoints"](#).

29.4 Using the ADF Logger and Business Component Browser

If you are not able to easily find the error in either your web page or its corresponding page definition file, you can use the JDeveloper debugging tools to investigate where your application failure occurs.

You can set up the Java logger to display Java diagnostic messages. You can set several levels of logging to control the level and number of messages that are displayed.

You can then use the ADF Declarative Debugger to set breakpoints and examine the internals of the application. For more information, see [Section 29.6, "Setting ADF Declarative Breakpoints"](#) and [Section 29.7, "Setting Java Code Breakpoints"](#).

29.4.1 How to Turn On Diagnostic Logging

Even before you use the actual debugger, running the application with framework diagnostics logging turned on can be helpful to see what happens when the problem occurs. To turn on diagnostic logging, set the Java system property named `jbo.debugoutput` to the value `console`. Additionally, the value `ADFLLogger` lets you route diagnostics through the standard `Logger` implementation, which can be controlled in a standard way through the `logging.xml` file.

The easiest way to set this system property while running your application inside JDeveloper is to edit your project properties and in the Run/Debug page, select a run configuration and click **Edit**. Then add the string `-Djbo.debugoutput=console` to the **Java Options** field.

29.4.2 How to Create an Oracle ADF Debugging Configuration

ADF Faces leverages the Java Logging API (`java.util.logging.Logger`) to provide logging functionality when you run a debugging session. Java Logging is a standard API that is available in the Java Platform at <http://java.sun.com/j2se/1.4.2/docs/guide/util/logging/overview.html>.

Because standard Java Logging is used, you can edit the `logging.xml` file to control the level of diagnostics you receive in the Log window.

To edit ADF package-level logging in the `logging.xml` file:

If you want to change the logging level for Oracle ADF, you can edit the `<logger>` elements of the `logging.xml` configuration file.

By default the level is set to `INFO` for all packages of Oracle ADF. Set `level="FINE"` for detailed logging diagnostics.

For the Oracle ADF view layer packages `oracle.adf.view.faces` and `oracle.adfinternal.view.faces`, edit:

```
<logger name="oracle.adf" level="FINE"/>
<logger name="oracle.adfinternal" level="FINE"/>
```

For the Oracle ADF Model layer packages, edit these elements:

```
<logger name="oracle.adf" level="FINE"/>
<logger name="oracle.jbo" level="FINE"/>
```

For the Oracle ADF Controller layer packages, edit these elements:

```
<logger name="oracle.adf.controller" level="FINE"/>
<logger name="oracle.adfinternal.controller" level="FINE"/>
```

Alternatively, you can create a debug configuration in JDeveloper that you can choose when you start a debugging session.

[Example 29–2](#) shows the portion of the `logging.xml` file where you can change the granularity of the log messages. Note in the example that the log for `oracle.adf.faces` has been changed to `FINE` to display more messages.

Example 29–2 Sample Section of the `logging.xml` Configuration File

```
</logging_configuration>
...
<loggers>
    <logger name="oracle.adf" level="INFO"/>
    <logger name="oracle.adf.faces" level="FINE"/>
    <logger name="oracle.adf.controller" level="INFO"/>
    <logger name="oracle.bc4j" level="INFO"/>
    <logger name="oracle.adf.portal" level="INFO"/>
    <logger name="oracle.vcr" level="INFO"/>
    <logger name="oracle.portlet" level="INFO"/>
    <logger name="oracle.adfinternal" level="INFO"/>
    <logger name="oracle.adfdt" level="INFO"/>
    <logger name="oracle.adfdtinternal" level="INFO"/>
</loggers>
</logging_configuration>
```

For the latest information about the different levels of the Java Logging system, go to the Sun Java website. Normally, the Java logging systems supports the following levels:

- SEVERE
- WARNING
- INFO
- CONFIG
- FINE
- FINER
- FINEST

To create an Oracle ADF Model debugging configuration:

1. In the Application Navigator, double-click the user interface project.
2. In the Project Properties dialog, click the **Run/Debug/Profile** node and create a new run configuration.

3. In the **Run Configurations** list, double-click the new run configuration to edit its properties.
4. In the Edit Run Configuration dialog, for **Launch Settings**, enter the following Java options for the default virtual machine:
`-Djbo.debugoutput=adflogger -Djbo.adflogger.level=FINE`
Set the `level=FINE` for detailed diagnostic messages.

To create an Oracle ADF view Javascript logging configuration:

1. In the Application Navigator, open the application or project `web.xml` file using the source editor.
2. Add the following elements to the file:

```
<context-param>
  <param-name>
    oracle.adf.view.rich.LOGGER_LEVEL
  </param-name>
  <param-value>
    FINE
  </param-value>
</context-param>
```

To create a core JSF Reference Implementation logging configuration:

1. Open the `logging.properties` file.

2. Set the following line to `FINE`:

```
<java.util.logging.ConsoleHandler.level=FINE
```

3. Add the following line:

```
com.sun.faces.level=FINE
```

The log now includes JSF Reference Implementation messages such as:

```
FINEST: End execute(phaseId=RESTORE_VIEW 1)
```

```
com.sun.faces.lifecycle.LifecycleImpl
haspostDataOrQueryParams
```

```
FINEST: Request Method: POST/PUT
```

```
com.sun.faces.lifecycle.LifecycleImpl execute
```

29.4.3 How to Use the Business Component Browser with the Debugger

Often you will find it useful to analyze and debug ADF Business Components without having to run the Fusion web application from within JDeveloper. You can use the ADF Business Component Tester tool to complement your debugging process. The tester allows you to view, insert, and update the contents of business objects, to add view criteria, and to set named bind variables. For more information, see [Section 6.3, "Testing View Object Instances Using the Business Component Browser"](#).

To launch the Business Component Tester:

1. In the Application Navigator, right-click the desired application module and choose **Run**.
2. In the Oracle Business Component Browser dialog, review the connection information and click **Connect** to launch the tester.

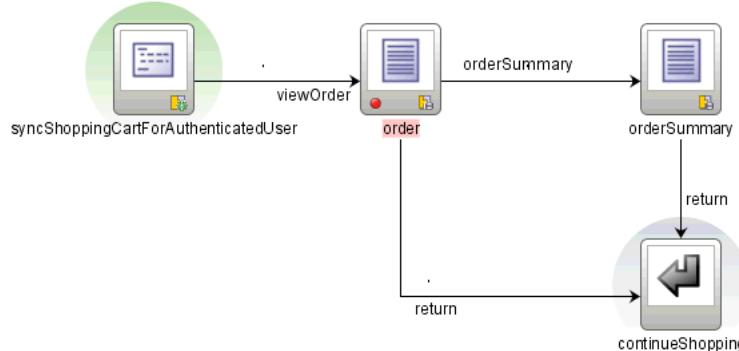
To launch the Business Component Tester and go into debug mode:

1. In the Application Navigator, select the desired application module and choose **Debug**.
2. In the Oracle Business Component Browser dialog, review the connection information and click **Connect** to launch the tester and go into debug mode.

29.5 Using the ADF Declarative Debugger

The ADF Declarative Debugger provides declarative breakpoints that you can set at the ADF object level (such as task flows, page definition executables, method and action bindings), as well as standard Java breakpoints. ADF declarative breakpoints provide a high-level object view for debugging ADF applications. For example, you can break before a task flow activity to see what parameters would be passed to the task flow, as shown in [Figure 29–3](#). To perform the same function using only Java breakpoints would require you to know which class or method to place the breakpoint in. ADF declarative breakpoints should be the first choice for ADF applications.

Figure 29–3 ADF Declarative Breakpoint on a Task Flow Activity

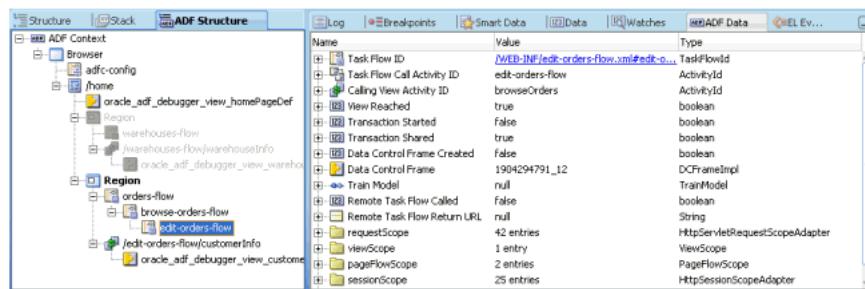


The ADF Declarative Debugger also supports standard Java code breakpoints. You can set these breakpoints when an ADF declarative breakpoint does not break in the place you want, either in addition to ADF breakpoints or when you are not debugging an ADF application.

The ADF Declarative Debugger is built on top of the Java debugger, so it has the features and behaviors of the Java debugger. But instead of needing to know the Java class or method, you can set ADF declarative breakpoints in visual editors.

When any breakpoint of either type is hit, you can examine the application status using a variety of windows. You can check where the break occurs in the Breakpoint window. You can check the call stack for the current thread using the Stack window. When you select a line in the Stack window, information in the Data window, Watches window, and all Inspector windows are updated to show relevant data. You can use the Data window to display information about arguments, local variables, and static fields in your application.

The ADF Structure window displays the runtime structure of the project. The ADF Data window automatically changes its display information based on the selection in the ADF Structure window. For example, if a task flow node is selected, the ADF Data window displays the relevant debugging information for task flows, as shown in [Figure 29–4](#).

Figure 29–4 ADF Structure Window and ADF Data Window for a Task Flow Selection

You can mix ADF declarative breakpoints with Java code breakpoints as needed in your debugging session. Although you can use Step functions to advance the application from Java code breakpoint to Java code breakpoint, the Step functions for ADF declarative breakpoints have more constraints and limitations. For more information about using Step functions on ADF declarative breakpoints, see [Table 29–3](#).

For information on how to use ADF declarative breakpoints, see [Section 29.6, "Setting ADF Declarative Breakpoints"](#).

For information on how to use Java breakpoints on classes and methods, see [Section 29.7, "Setting Java Code Breakpoints"](#)

In a JSF application (including ADF Fusion applications), when a breakpoint breaks, you can use the EL Evaluator to examine the value of an EL expression. The EL Evaluator has the browse function that helps you select the correct expression to evaluate. For more information, see [Section 29.5.4, "How to Use the EL Expression Evaluator"](#).

Whether you plan to use ADF declarative breakpoints or Java breakpoints, you can use the ADF Declarative Debugger with Oracle ADF source code. You can obtain Oracle ADF source code with Debug libraries. For more information about loading source code. see [Section 29.5.1, "Using ADF Source Code with the Debugger"](#).

29.5.1 Using ADF Source Code with the Debugger

If you have valid Oracle ADF support, you can obtain complete source code for Oracle ADF by opening a service request with Oracle Worldwide Support. You can request a specific version of the Oracle ADF source code. You may be given download and password information to decrypt the source code ZIP file. Contact Oracle Worldwide Support for more information.

Adding Oracle ADF source code access to your application debugging session will:

- Provide access to the JDeveloper Quick Javadoc feature in the source editor. Without the source code, you will have only standard Javadoc.
- Enhance the use of Java code breakpoints by displaying the Oracle source code that's being executed when the breakpoint is encountered. You can also set breakpoints easier by clicking on the margin in the source code line you want to break on. Without the source code, you will have to know the class, method, or line number in order to set a breakpoint within Oracle code.
- For Java code breakpoints, you will be able to see the values of all local variables and member fields in the debugger.

The ADF source code ZIP file may be delivered within an encrypted "outer" ZIP file to protect its contents during delivery. The "outer" ZIP name is sometimes a variant of the service request number.

After you have received or downloaded the "outer" ZIP, unzip it with the provided password to access the actual source code ZIP file. The ADF source code ZIP name should be a variant of the ADF version number and build number. For example, the ADF source ZIP may have a format similar to `adf_vvvv_nnnn_source.zip`, where `vvvv` is the version number and `nnnn` is the build number.

After you have access to the source code ZIP, extract its contents to a working directory.

29.5.2 How to Set Up the ADF Source User Library

You create a name for the source user library and then associate that name with the source zip file.

To add the ADF source zip file to the user library

1. From the main menu, choose **Tools > Manage Libraries**.
2. In the Manage Libraries dialog, with the **Libraries** tab selected, click **New**.
3. In the Create Library window, enter a library name for the source that identifies the type of library.
4. Select the **Source Path** node in the tree structure. Click **Add Entry**.

Note: Do not enter a value for the class path. You need to provide a value only for the source path.

5. In the Select Path Entry window, browse to the directory where the file was extracted and select the source zip file. Click **Select**.
6. In the Create Library window, verify that the source path entry has the correct path to the source zip file, and deselect **Deployed by Default**. Click **OK**.
7. Click **OK**.

29.5.3 How to Add the ADF Source Library to a Project

After the source library has been added to the list of available user libraries, add it to the project you want to debug.

To add the ADF source zip file to the project:

1. In the Application Navigator, double-click the project or right-click the project and select **Project Properties**.
2. In the Project Properties dialog, select **Libraries and Classpaths**.
3. Click **Add Library**.
4. In the Add Library dialog, under the **Users** node, select the source library you want to add and click **OK**.

The source library should appear in the **Classpath Entries** section in the Project Properties dialog.

5. Click **OK**.

29.5.4 How to Use the EL Expression Evaluator

JDeveloper provides an EL expression evaluator that allows you to enter an EL expression for evaluation. You can enter arbitrary EL expressions for evaluation within the current context. If the EL expression no longer applies within the current context, the value will be evaluated to null.

The EL Evaluator is different from the Watches window in that EL evaluation occurs only when stopped at a breakpoint, not when stopped at subsequent debugging steps.

The EL Evaluator is available for debugging any JSF application.

Caution: Be wary when you are evaluating EL expressions that you do not inadvertently change application data and therefore the behavior of the application.

To use the EL Evaluator:

1. Set a breakpoint in the application.

The application must be a JSF application. It does not need to be an ADF application.

2. Start the debugging process.

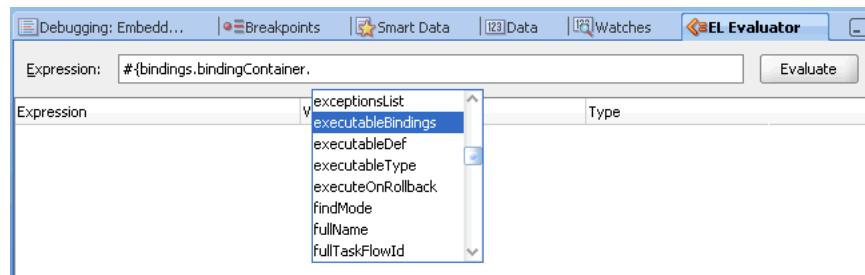
You can:

- From the main menu, choose **Run > Debug**.
 - From the Application Navigator, right-click the project, `adfc-config.xml`, `faces-config.xml`, task flow, or page and choose **Debug**.
 - From the task flow editor, right-click an activity and choose **Debug**. Only task flows that do not use page fragments can be run.
3. When the breakpoint is reached, the EL Evaluator should appear as a tab in the debugger window area. Click the **EL Evaluator** tab to bring it forward. If it does not appear, choose **View > Debugger > EL Evaluator** from the main menu.

Note: Be sure that the application has actually hit a breakpoint by checking the Breakpoint window or checking that there is an **Execution Point** icon (red right arrow) next to the breakpoint. Depending on where you set the breakpoint, an application may appear to be stopped when in fact it is waiting for user input at the page.

4. Enter an EL expression in the input field.

When you click in the field after entering `# {` or after a period, a discovery function provides a selectable list of expression items, as shown in [Figure 29–5](#). Auto-completion will be provided for easy entry. You can evaluate several EL expressions at the same time by separating them with semicolons.

Figure 29–5 Using the Discovery Function of the EL Evaluator

5. When you finish entering the EL expression, click **Evaluate** and the expression is evaluated, as shown in Figure 29–6.

Figure 29–6 EL Expression Evaluated

Expression	Value	Type
#{bindings.bindingContainer.executableBindings}	ArrayList<java.lang.Object>	
#{bindings.bindingContainer.executableBindings[0]}	data.oracle_adf_debugger_view_homeP... JUFormBinding	
#{bindings.bindingContainer.executableBindings[1]}	data.oracle_adf_debugger_view_homeP... JUFormBinding	
#{bindings.bindingContainer.executableBindings[2]}	data.oracle_adf_debugger_view_homeP... JUFormBinding	
#{bindings.bindingContainer.mParamsList}	1 elements	ArrayList<java.lang.Object>
#{bindings.bindingContainer.mExecutablesList}	1 elements	ArrayList<java.lang.Object>
#{bindings.bindingContainer.mExecutablesList[0]}	CustomersView1Iterator	JUIteratorBinding
#{bindings.bindingContainer.mExecutablesList[0].mName}	"CustomersView1Iterator"	String
#{bindings.bindingContainer.mExecutablesList[0].mSourceName}	"CustomersView1"	String
#{bindings.bindingContainer.mExecutablesList[0].mDC}		JUApplication
#{bindings.bindingContainer.mExecutablesList[0].mBC}	data.oracle_adf_debugger_view_homeP... JUFormBinding	
#{bindings.bindingContainer.mExecutablesList[0].mRSI}		ViewObjectImpl
#{bindings.bindingContainer.mExecutablesList[0].mVO}		ViewObjectImpl

29.5.5 How to View and Export Stack Trace Information

If you are unable to determine what the problem is and to resolve it yourself, typically your next step is to ask someone else for assistance. Whether you post a question in the OTN JDeveloper Discussion Forum or open a service request on Metalink, including the stack trace information in your posting is extremely useful to anyone who will need to assist you further to understand exactly where the problem is occurring.

JDeveloper's Stack window makes communicating this information easy. Whenever the debugger is paused, you can view the Stack window to see the program flow as a stack of method calls that got you to the current line. Right-click the Stack window background and choose **Preferences**. You can set the Stack window preference to include the line number information, as well as the class and method name that will be there by default. Finally, the context menu option **Export** lets you save the current stack information to an external text file whose contents you can then post or send to whomever might need to help you diagnose the problem.

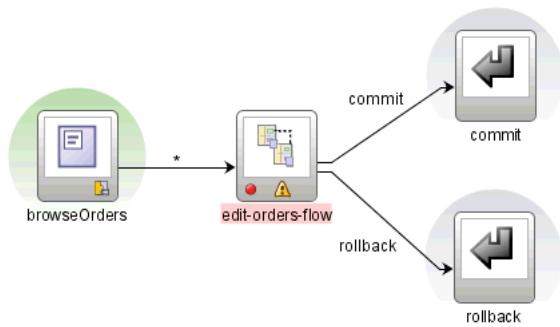
29.6 Setting ADF Declarative Breakpoints

You use the ADF Declarative Debugger features in JDeveloper to declaratively set breakpoints on ADF task flow activities, page definition executables, and method, action, and value bindings. Instead of needing to know all the internal constructs of the ADF code, such as method names and class names, you can set breakpoints at the highest level of object abstraction.

You can add breakpoints to task flow activities in the **Diagram** tab of the task flow editor. You can select a task flow activity and use the context menu to toggle or disable breakpoints on that activity, or press the F5 button. After the application pauses at the breakpoint, you can view the runtime structure of the objects in the ADF Structure window as a tree structure. The ADF Data window displays a list of data for a given object selected in the ADF Structure window.

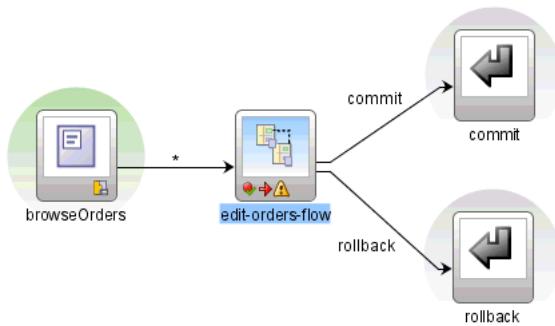
For example, when you set a breakpoint on view activity in the Orders task flow, a red dot icon appears in the view activity, as shown in [Figure 29–7](#).

Figure 29–7 ADF Declarative Breakpoint on a Task Flow Activity

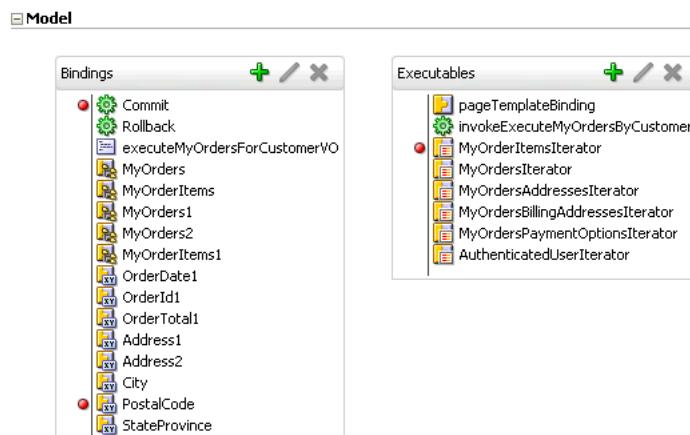


When the breakpoint is reached, the application is paused and the icon changes, as shown in [Figure 29–8](#).

Figure 29–8 Application Paused at an ADF Declarative Breakpoint



Similarly, you can set ADF declarative breakpoints in the page definition file. You set breakpoints for items in the bindings and executables lists using the context menu. Again, a red dot icon that indicates the breakpoint is set, as shown in [Figure 29–9](#).

Figure 29–9 ADF Declarative Breakpoints in the Page Definition File

You can define both ADF declarative breakpoints and standard Java code breakpoints when using the ADF Declarative Debugger. Depending on your debugging scenario, you may only need to use the declarative breakpoints to debug the application. Or you may find it necessary to add additional breakpoints in Java code that are not available declaratively. For information on Java code breakpoints, see [Section 29.7.1, "How to Set Java Breakpoints on Classes and Methods"](#). Table 29–1 lists the ADF declarative debugger breakpoints.

Table 29–1 ADF Declarative Debugger Breakpoints

ADF Area	Declarative Breakpoint	JDeveloper Editor	JDeveloper Location	JDeveloper Context Menu Command	Description
Page Definition	Before executable: <ul style="list-style-type: none">▪ Iterator▪ invokeAction▪ Region instantiation	Page definition editor	Page Definition Overview tab > Executables section > executable row	Toggle Breakpoint or F5	Pauses debugging when executable is refreshed. For task flow bindings, this represents two times per lifecycle: first, during prepareModel (initial region creation), and then again during prepareRender (where dynamic regions swap their corresponding task ID).
	Before action binding: <ul style="list-style-type: none">▪ methodAction▪ Built-in operations		Page Definition Overview tab > Bindings section > binding row	Toggle Breakpoint or F5	Pauses debugging when binding is executed.
	Before attribute value binding		Page Definition Overview tab > Bindings section > binding row	Toggle Breakpoint or F5	Pauses debugging before the attribute's setInputValue() ADF source code method is executed. New values will be the parameters to setInputValue().
ADF Task Flow	Before activity	ADF task flow editor	Task flow editor > Diagram tab > Activity node	Toggle Breakpoint or F5	Pauses debugging before the activity executes. The activity where the declarative breakpoint is defined has not yet been performed. View activities provide the additional ability to "Step Into" the view activity to pause when the corresponding page or page fragment's component tree is available for inspection in the ADF Data window.

Table 29–2 lists where and how an ADF declarative breakpoint will appear in the Breakpoint window. When you double-click an ADF declarative breakpoint in the Breakpoint window, the corresponding object editor will appear in the workspace showing the breakpoint location.

Table 29–2 Breakpoint Window Display of Declarative Breakpoints

Declarative Breakpoint Type	Breakpoint Description Column	Breakpoint Type Column
Before page definition executable:	Before <i>page definition@executable id</i>	Page definition executable breakpoint
<ul style="list-style-type: none"> ▪ Iterator ▪ invokeAction ▪ Region instantiation ▪ 		
Before page definition action binding:	Before <i>page definition@binding id</i>	Page definition binding breakpoint
<ul style="list-style-type: none"> ▪ methodAction ▪ Built-in Operations 		
Before page definition attribute value binding	Before <i>page definition@binding id</i>	Page definition binding breakpoint
Before ADF task flow activity	Before <i>task flow document#task flow id@activity id</i>	Task flow activity breakpoint

Table 29–3 lists the Step commands that can be used with ADF declarative breakpoints.

Table 29–3 ADF Declarative Debugger Step Commands

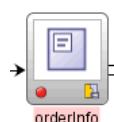
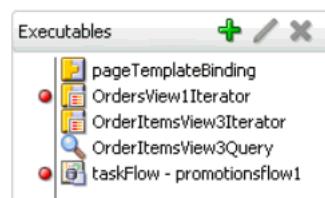
ADF Debugger Step Commands	Description
Find Execution Point	Supported for declarative breakpoints to display the current execution point open and active within the corresponding editor.
Step Over (F8)	Supported for task flow activity declarative breakpoints to step from activity to activity within a task flow. If user interaction is required (for example, page displayed), once it is received (for example, button selected), processing will resume and then will pause before the next task flow activity.
	Supported also for all page definition declarative breakpoints to step to the next page definition executable (for example, iterators, invokeAction) to be performed, if any. Stepping to the next individual binding will not be supported.

Table 29–3 (Cont.) ADF Declarative Debugger Step Commands

ADF Debugger Step Commands	Description
Step Into (F7)	Supported only for task flow activity declarative breakpoints defined on view activities and task flow call activities. Task flow activity declarative breakpoints pause debugging just before the activity is executed. Performing a “Step Into” provides the ability to pause debugging at a point after the view activity or task flow call activity is executed. For view activity declarative breakpoints, Step Into allows debugging to continue until just after the view activity is executed, but before the new page is rendered. The ADF Structure and ADF Data windows will update to reflect the new page about to be rendered, allowing you to inspect the state for the corresponding view activity.
	For task flow call activity declarative breakpoints, debugging will continue until just prior to executing the called task flow default activity. This action would be the same as placing a task flow activity declarative breakpoint on the called task flow default activity.
Step Out (Shift F7)	Supported for task flow activity declarative breakpoints to step out of the current called task flow and back into the caller (if any). If user interaction is required (for example, page displayed) once user interaction received (for example, button selected), processing will resume and will pause before the next user interaction or activity within the calling task flow.
Continue Step (Shift F8)	Not supported for declarative breakpoints.
Step to End of Method	Not supported for declarative breakpoints.
Run to Cursor	Not supported for declarative breakpoints.
Pop Frame	Supported for declarative breakpoints, as it is for Java code, to return to a previous point of execution.

The ADF Declarative Debugger uses the standard debugger icons and notations for setting, toggling, and indicating the status of ADF declarative breakpoints.

When an ADF declarative breakpoint is set, it appears as a red dot icon in the task flow activity or in the page definition breakpoint margin, as shown in [Figure 29–10](#) and [Figure 29–11](#).

Figure 29–10 ADF Declarative Breakpoint Enabled on a Task Flow Activity**Figure 29–11 ADF Declarative Breakpoints Enabled in the Page Definition Executables**

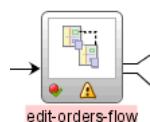
When an ADF declarative breakpoint is disabled, the red icon becomes a gray icon, as shown in [Figure 29–12](#).

Figure 29–12 ADF Declarative Breakpoint Disabled



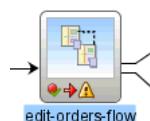
When an ADF declarative breakpoint is active, the red dot icon has a green checkmark, as shown in [Figure 29–13](#).

Figure 29–13 ADF Declarative Breakpoint Active



When the application is paused at an ADF declarative breakpoint, an **Execution Point** icon appears to the right of the breakpoint icon, as shown in [Figure 29–14](#).

Figure 29–14 Application Paused at an Execution Point



29.6.1 How to Set Task Flow Activity Breakpoints

After you have created a task flow diagram, you can set ADF declarative breakpoints on task flow activities.

To set a breakpoint on a task flow activity:

1. Open the task flow in the task flow editor **Diagram** tab.
 2. Select the task flow activity node, right-click and choose **Toggle Breakpoint** from the context menu, or press F5.
- A breakpoint icon appears on the task flow activity.
3. Start the debugging process.

You can:

- From the main menu, choose **Run > Debug**.
- From the Application Navigator, right-click the project, `adf-config.xml`, `faces-config.xml`, task flow, or page and choose **Debug**.
- From the task flow editor, right-click an activity and choose **Debug**. Only task flows that do not use page fragments can be run.

4. When the application is paused, an **Execution Point** icon appears next to the breakpoint icon on the task flow activity. You can examine the application using different debugger windows.

Note: Be sure that the application has actually hit a breakpoint by checking the Breakpoint window or checking that there is an **Execution Point** icon (red right arrow) next to the breakpoint.

Depending on where you set the breakpoint, an application may appear to be stopped when in fact it is waiting for user input at the page.

The application is paused before the task flow activity executes.

5. The ADF Structure window and the ADF Data window appear by default, as well as several debugger windows. You can examine the runtime structure in the ADF Structure window and its corresponding data in the ADF Data window. See [Section 29.6.5, "How to Use the ADF Structure Window"](#) and [Section 29.6.6, "How to Use the ADF Data Window"](#).
6. Select a node in the ADF Structure window and view pertinent information in the ADF Data window. If the breakpoint is on a task flow view activity, the breakpoint occurs before the activity is performed, and so the ADF Structure window will display the current state of the application just prior to the view activity. Use the Step Into (F7) function to step further into the application lifecycle after the view activity is performed.
7. Continue debugging the application as required, using the Step functions as described in [Table 29–3](#). The key Step functions are Step Into (F7) and Step Over (F8).

When the application is paused, you can remove or disable existing breakpoints and set new breakpoints.

29.6.2 How to Set Page Definition Executable Breakpoints

If your page definition has executables, you can set breakpoints to pause the application on these executables. For example, you can set breakpoints to pause the application when iterators are refreshed or when `invokeAction` methods are performed.

To set a breakpoint on an executable in the page definition file:

1. Open the page definition file editor. Select the **Overview** tab.
 2. In the overview editor, select an executable from the **Executables** list and click in the breakpoint margin to the left of the item.
- A breakpoint icon appears in the margin next to the item.
3. Start the debugging process.

You can:

- From the main menu, choose **Run > Debug**.
- From the Application Navigator, right-click the project, `adf-config.xml`, `faces-config.xml`, task flow, or page and choose **Debug**.
- From the task flow editor, right-click an activity and choose **Debug**. Only task flows that do not use page fragments can be run.

4. When the application is paused, an **Execution Point** icon appears in the margin next to the breakpoint icon of the executable item. You can examine the application using several debugger windows.

The application pauses when the executable binding is refreshed. If this is a task flow executable, the pause occurs in the `prepareModel` and the `prepareRender` lifecycles.

Note: Be sure that the application has actually hit a breakpoint by checking the Breakpoint window or checking that there is an **Executable Point** icon (red right arrow) next to the breakpoint. Depending on where you set the breakpoint, an application may appear to be stopped when in fact it is waiting for user input at the page.

5. The ADF Structure window and the ADF Data window appear by default, as well as several debugger windows. You can examine the runtime structure in the ADF Structure window and its corresponding data in the ADF Data window. See [Section 29.6.5, "How to Use the ADF Structure Window"](#) and [Section 29.6.6, "How to Use the ADF Data Window"](#).
6. Select a node in the ADF Structure window and view pertinent information in the ADF Data window.
7. Continue debugging the application as required, using the Step functions as described in [Table 29–3](#). The key Step function is Step Over (F8).

When the application is paused, you can remove or disable existing breakpoints and set new breakpoints.

29.6.3 How to Set Page Definition Action Binding Breakpoints

You can set breakpoints in the page definition file on action bindings and `methodAction` bindings. The application pauses when the binding is executed.

To set a breakpoint on an action binding in the page definition file:

1. Open the page definition file editor. Select the **Overview** tab
2. Select a `methodAction` binding or built-in operation item from the **Bindings** list and click in the breakpoint margin to the left of the item.

A breakpoint icon appears next to the item.

3. Start the debugging process.

You can:

- From the main menu, choose **Run > Debug**.
 - From the Application Navigator, right-click the project, `adfc-config.xml`, `faces-config.xml`, task flow, or page and choose **Debug**.
 - From the task flow editor, right-click an activity and choose **Debug**. Only task flows that do not use page fragments can be run.
4. When the application pauses, an **Execution Point** icon appears next to the breakpoint icon on the action binding item. You can examine the application using several debugger windows.

The application is paused when the binding is executed.

Note: Be sure that the application has actually hit a breakpoint by checking the Breakpoint window or checking that there is an **Execution Point** icon (red right arrow) next to the breakpoint. Depending on where you set the breakpoint, an application may appear to be stopped when in fact it is waiting for user input at the page.

5. The ADF Structure window and the ADF Data window appear by default, as well as several debugger windows. You can examine the runtime structure in the ADF Structure window and its corresponding data in the ADF Data window. See [Section 29.6.5, "How to Use the ADF Structure Window"](#) and [Section 29.6.6, "How to Use the ADF Data Window"](#).
6. Select a node in the ADF Structure window and view pertinent information in the ADF Data window.
7. Continue debugging the application as required, using the Step functions as described in [Table 29–3](#). The key Step function is Step Over (F8).

When the application is paused, you can remove or disable existing breakpoints and set new breakpoints.

29.6.4 How to Set Page Definition Attribute Value Binding Breakpoints

If the page definition has attribute values bindings, you can set breakpoints on the attribute value bindings to pause the application.

To set a breakpoint on an attribute value binding in the page definition file:

1. Open the page definition file editor. Select the **Overview** tab.
2. Select an attribute value from the **Bindings** list and click on the breakpoint margin to the left of the item. A breakpoint icon appears next to the attribute value binding.
3. Start the debugging process.

You can:

- From the main menu, choose **Run > Debug**.
- From the Application Navigator, right-click the project, `adfc-config.xml`, `faces-config.xml`, task flow, or page and choose **Debug**.
- From the task flow editor, right-click an activity and choose **Debug**. Only task flows that do not use page fragments can be run.

4. When the application is paused, an **Execution Point** icon appears next to the breakpoint icon on the attribute value binding. You can examine the application using several debugger windows.

The application is paused before the `setInputValue()` method of the ADF source code. New values will be the parameters that go into this method.

Note: Be sure that the application has actually hit a breakpoint by checking the Breakpoint window or checking that there is an **Execution Point** icon (red right arrow) next to the breakpoint. Depending on where you set the breakpoint, an application may appear to be stopped when in fact it is waiting for user input at the page.

5. The ADF Structure window and the ADF Data window appear by default, as well as several debugger windows. You can examine the runtime structure in the ADF Structure window and its corresponding data in the ADF Data window. See [Section 29.6.5, "How to Use the ADF Structure Window"](#) and [Section 29.6.6, "How to Use the ADF Data Window"](#).
6. Select a node in the ADF Structure window and view pertinent information in the ADF Data window.
7. Continue debugging the application as required, using the Step functions as described in [Table 29–3](#). The key Step function is Step Over (F8).

When the application is paused, you can remove or disable existing breakpoints and set new breakpoints.

29.6.5 How to Use the ADF Structure Window

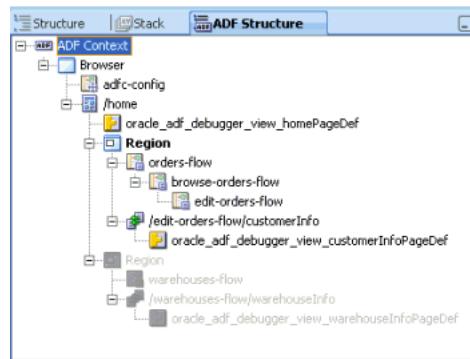
When the application is paused at a breakpoint, the ADF Structure window displays a tree structure of the ADF runtime objects and their relationships within the application. In particular, it shows the hierarchy of view ports, which represent either the main browser window or contained regions and portlets. When you select different items in the ADF Structure window, the data display in the accompanying ADF Data window changes. For more information about the ADF Data window, see [Section 29.6.6, "How to Use the ADF Data Window"](#).

The ADF Structure window and the ADF Data window are shown by default during a debugging session when either of the following is true:

- The project being debugged contains a WEB-INF/adfc-config.xml file.
- The project being debugged contains any ADF Faces tag libraries.

You can launch the ADF Structure window by choosing **View > Debugger > ADF Structure** from the main menu.

When a breakpoint is encountered, the ADF Structure window displays a tree structure of the runtime objects, as shown in [Figure 29–15](#).

Figure 29–15 ADF Structure Window

When you select an item in the ADF Structure window, the data and values associated with that item are displayed in the ADF Data window. [Figure 29–16](#) shows a task flow selected in the ADF Structure window, with its corresponding information displayed in the ADF Data window.

Figure 29–16 ADF Structure Window Selection and ADF Data Window Data

The screenshot shows the ADF Structure window on the left and the ADF Data window on the right. The ADF Structure window displays the same tree structure as Figure 29–15. The ADF Data window has tabs for 'Log', 'Breakpoints', and 'Smart...', with the 'Breakpoints' tab selected. It shows a table with columns 'Name', 'Value', and 'Type'. The table contains the following data:

Name	Value	Type
Task Flow ID	/WEB-INF/browse-ord...	TaskFlowId
Task Flow Call Activity ID	browse-orders-flow	ActivityId
Calling View Activity ID	null	ActivityId
View Reached	true	boolean
Transaction Started	true	boolean
Transaction Shared	false	boolean
Data Control Frame Create	false	boolean
Data Control Frame	293352050_1	DCFrameImpl
Train Model	null	TrainModel
Remote Task Flow Called	false	boolean
Remote Task Flow Return	Null	String
requestScope	45 entries	HttpServletRequest5c...

The root of the hierarchy is **ADF Context**. The current view port where processing has stopped appears in bold. Default selections within the tree will be retained from the previous breakpoint, so you can monitor any changes between breakpoints. The ADF object where the ADF declarative breakpoint was defined will be opened in the corresponding JDeveloper editor, either the task flow editor or the page definition editor.

The ADF Structure tree will be rebuilt each time the application breaks and at subsequent steps to reflect the changed state of the objects. Although the entire tree hierarchy will be displayed, only items within the current view port and its parent view port(s) will be available for selection and further inspection. All other items in the tree hierarchy not in the current context will be dimmed and disabled. You can still use the hierarchy to identify runtime object relationships within the application, but it will be limited to the current context (and its parent view ports).

[Table 29–4](#) lists the different types of items that can be displayed in the ADF Structure window hierarchy tree.

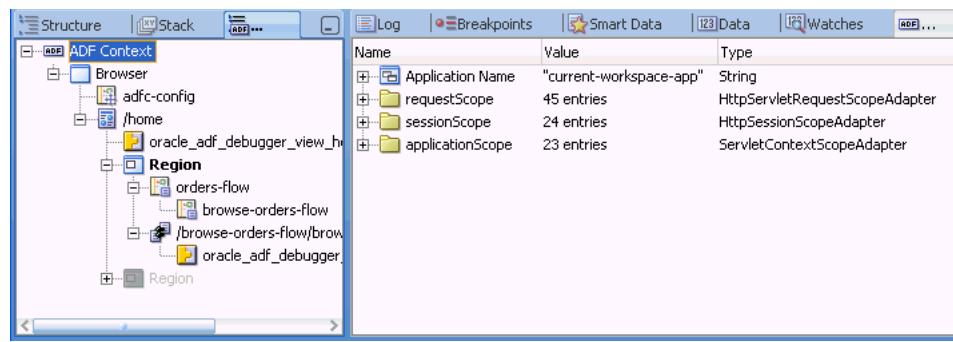
Table 29–4 ADF Structure Window Items

ADF Structure Tree Item	Description
ADF context	Always displayed as the single root of the ADF Structure hierarchy. There will only be one ADF Context within the ADF Structure hierarchy.
View port	<p>View ports are an ADF Controller concept. For this reason, view ports appear within the ADF Structure hierarchy only when the application being debugged utilizes ADF Controller.</p> <p>View ports can represent one of the following:</p> <ul style="list-style-type: none"> ▪ Browser: Main browser view ports appear as children of the root ADF Context. If multiple browser windows are open during the debugging runtime session, multiple browser view ports are presented within the hierarchy. The label of each browser view port displays the text "Browser". The view port also provides a tooltip for the view port ID similar to the following example: "View Port: 999999". ▪ Region: Region view ports appear as the children of page or page fragments. The label of each region view port displays the text "Region". The region also provides a tooltip for the view port ID similar to the following example: "View Port: 999999".
ADF task flows	<p>The task flow call stack corresponding to each view port appears as a hierarchy of ADF task flows. The initial ADF task flow called for the stack is a direct child of its corresponding view port. The label of each ADF task flow reflects the corresponding ADF task flow display name (if any) or its task flow ID. Region view ports will not display the item in their task flow call stack hierarchy for their implied unbounded task flow. The task flow also provides a tooltip displaying the ADF task flow path, and a context menu item to open to the corresponding ADF task flow within the editor workspace.</p> <p>If ADF Controller is not utilized in the application (or if the page is run outside the context of an ADF task flow), ADF task flows will not appear within the hierarchy.</p>
Page	Represents the page (view) currently displayed within a browser or portlet view port. Presented along with its associated page definition (if any) as a child. If the application being debugged utilizes ADF Controller, pages will be children of each browser or portlet view port. The label of each page reflects its corresponding runtime view ID. The page also provides a tooltip displaying the page path, and a context menu item to open to the corresponding page within the editor workspace. If a visual user interface is not implemented for the application, the page will not appear within the hierarchy.
Page fragment	Represents the page fragment currently displayed within a region view port. Presented along with its associated page definition (if any) as a child. If the application being debugged utilizes ADF Controller, page fragments will be children of each region view port. The label of each page fragment node reflects its corresponding runtime view ID. The page fragment also provides a tooltip displaying the page fragment path, and a context menu item to open to the corresponding page fragment within the editor workspace.
Binding container	Represents the binding container for the corresponding page or page fragment. The label of each binding container reflects its corresponding file name (page definition file) without the extension. The binding container also appears under current task flows when used to represent task flow activity bindings (for example, method call activity bindings).
	If ADF Model is not utilized for the application, binding containers will not appear.

29.6.6 How to Use the ADF Data Window

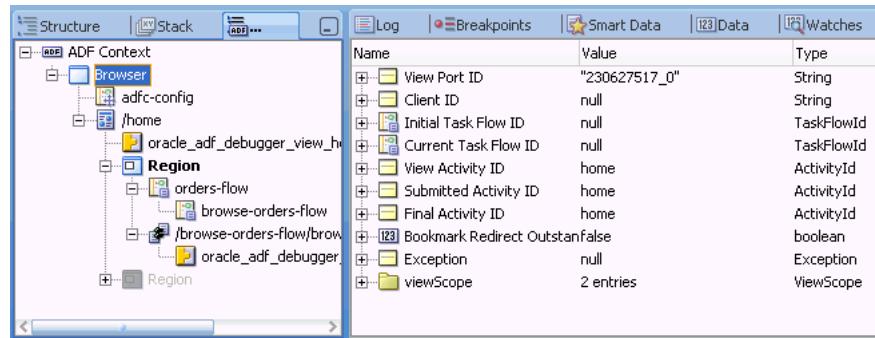
When an application is paused at an ADF declarative breakpoint, the ADF Data window displays relevant data based on the selection in the ADF Structure window. You can launch the ADF Data window by choosing **View > Debugger > ADF Data** from the main menu.

When the ADF context is selected in the ADF Structure window, as shown in [Figure 29–17](#), the application name and memory scopes will be displayed in the ADF Data window, as listed in [Table 29–5](#).

Figure 29–17 ADF Context Selected for ADF Data Window**Table 29–5 ADF Data Window Content for an ADF Structure Window Selection**

ADF Structure Window	ADF Data Content
ADF context	Displays the application name and content of the standard JSF scopes.
View port	Displays view port details, including the view scope contents.
Task flow	Displays information for the selected ADF task flow, including current transaction status and ADFm save point status.
Page/page fragment	Displays the page or page fragment UI component tree hierarchy for the selected page or page fragment if the page or page fragment has been rendered.
Binding container	Displays the binding container runtime values, including parameters, bindings, and events.

Selecting a view port within the ADF Structure hierarchy will display the view port's current view port details in the ADF Data window, as shown in [Figure 29–18](#). Values displayed for each view port are summarized in [Table 29–6](#).

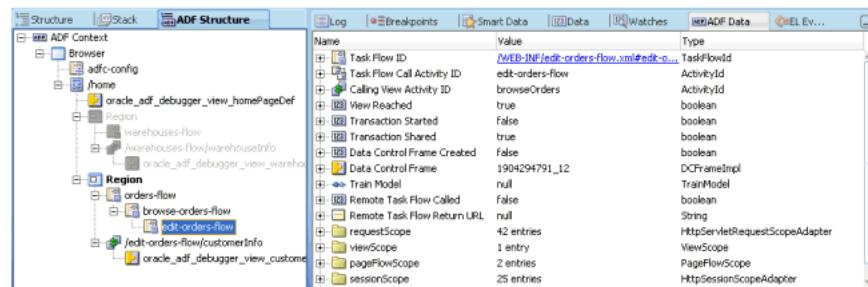
Figure 29–18 View Port Selected for ADF Data Window**Table 29–6 ADF Data Window Content for View Port**

View Port	Description
View port ID	Not displayed
Client ID	Not displayed
Initial task flow ID	Initial ADF task flow on the view ports task flow call stack. Appears as a link to open the corresponding task flow definition in the editor workspace.
Current task flow ID	Not displayed

Table 29–6 (Cont.) ADF Data Window Content for View Port

View Port	Description
View activity ID	Current ADF task flow view activity ID. Applicable only if the current ADF task flow activity is a view activity.
Submitted activity ID	ADF task flow activity submitting the current request.
Final activity ID	ADF task flow activity receiving the current request.
Bookmark redirect outstanding	(Boolean)
exception	(If any)
viewScope	Memory scope available from the view port context displaying its variables in a hierarchy.

In the ADF Structure window, each individual ADF task flow within a task flow call stack hierarchy is selectable. An ADF task flow selected in the ADF Structure window will display the current task flow information in the ADF Data window, as shown in [Figure 29–19](#). Task flow templates utilized by the selected ADF task flow will be determined by manually navigating to the ADF task flow source file. This is the same way similar functionalities are handled for Java source files. Current information for a selected ADF task flow is summarized in [Table 29–7](#).

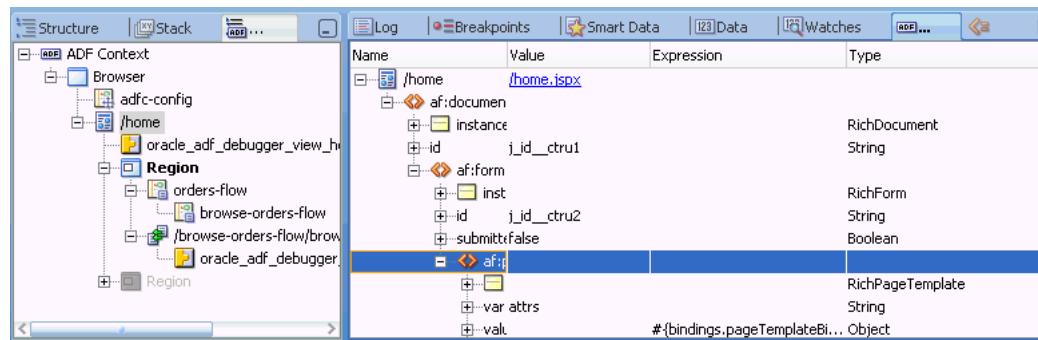
Figure 29–19 Task Flow Selected for ADF Data Window**Table 29–7 ADF Data Window Content for Task Flow**

Task Flow	Description
ADF task flow reference	Appears as a navigation link for bounded task flows to open the task flow definition source file within the editor workspace. Link text consists of <code>task flow source document#task flow id</code> .
Task flow call activity Id	Task flow activity ID for the calling task flow. Will be <code>null</code> for the first ADF task flow within each view port task flow call hierarchy.
Calling view activity Id	The calling view activity of the current view activity displayed by the ADF task flow, if any.
View reached	(Boolean)
Train model	Only applicable if the ADF task flow is created as a train.
Transaction started	(Boolean) Identifies the current status of the ADF task flow transactional state. For example, did the ADF task flow begin a new transaction?
Transaction shared	(Boolean) Identifies the current status of the ADF task flow transactional state. For example, did the ADF task flow join an existing transaction?

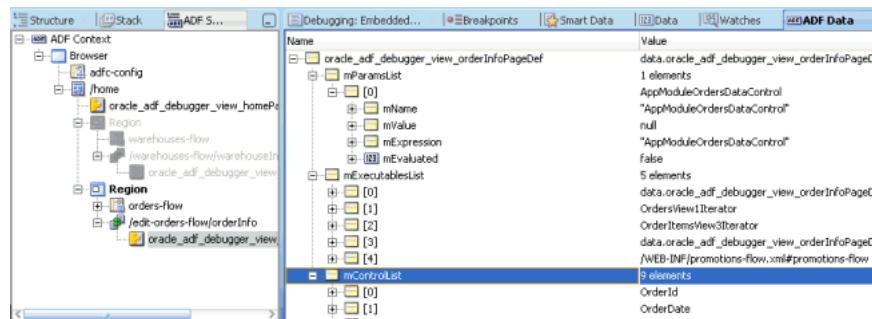
Table 29–7 (Cont.) ADF Data Window Content for Task Flow

Task Flow	Description
Save point	Identifies the current status of the ADF task flow's ADFm save point creation state. For example, was a model save point created upon ADF task flow entry?
Remote task flow called	(Boolean)
Remote task flow return URL	Applies only when calling an ADF task flow remotely. Identifies the URL for return once the task flow called remotely completes.
Data control frame created	(Boolean)
Data control frame	Name of data control frame associated with the ADF task flow.
Memory scopes	Supports viewing memory scopes such as request, view, page flow, application, session, and none. Each memory scope will be presented as a top-level, expandable object. The contents of the page flow and view scopes will be tied to the ADF task flow currently selected in the new ADF Structure window. When a different ADF task flows scope is selected in the ADF Structure window, the ADF page flow and view scope contents will change accordingly.

When you select a page or page fragment node in the ADF Structure hierarchy, the corresponding UI component tree is displayed within the ADF Data window, as shown in [Figure 29–20](#). If a page or page fragment is based on a page template, any content coming from the page template will appear outside any facet reference elements.

Figure 29–20 Page Selected for ADF Data Window

When you select a binding container in the ADF Structure hierarchy, it will display the current values of its corresponding binding container bindings within the ADF Data window, as shown in [Figure 29–21](#). These values are listed in [Table 29–8](#).

Figure 29–21 Binding Container Selected for ADF Data Window**Table 29–8 ADF Data Window Content for Binding Container**

Binding Container	Description
Page definition link	Navigates to the corresponding page definition source file and opens it within the editor workspace.
Parameters	Current values of all binding container parameters.
Iterator attribute bindings	Each iterator will list its corresponding attribute bindings.
Task flow bindings	Current value of the task flow id assigned to the taskFlow binding and all of its associated parameter values. Task flow IDs will appear as links navigating to open the corresponding task flow definition source file within the editor workspace. Link text consists of <code>task flow source document#task flow id</code> .
Binding container of page template	If the corresponding page or page fragment utilized a page template, the binding container of the page template will appear as a child of page or page fragment binding container content.

29.7 Setting Java Code Breakpoints

You can use the ADF Declarative Debugger to set breakpoints on Java classes and methods, as in any standard Java code debugger. You can use Java code breakpoints in combination with ADF declarative breakpoints or by themselves. For most ADF applications, ADF declarative breakpoints will provide enough debugging information to troubleshoot the application. For information about using ADF declarative breakpoints, see [Section 29.6, "Setting ADF Declarative Breakpoints"](#). However, you may need to set breakpoints on specific classes or methods for further inspection. Or, you may be debugging a non-ADF application, in which case, you can use Java code breakpoints.

JDeveloper provides a class locator feature that assists you in finding the class you want to break on. If you can obtain Oracle ADF source code, you can enhance your debugging by having access to various ADF classes and methods. For more information about getting ADF source code, see [Section 29.5.1, "Using ADF Source Code with the Debugger"](#). If you obtained the ADF source, you can further enhance the debugging experience by using the debug library version of the ADF source, as described in [Section 29.7.4, "How to Use Debug Libraries for Symbolic Debugging"](#).

29.7.1 How to Set Java Breakpoints on Classes and Methods

You can set Java breakpoints on your classes and methods. If you have ADF source code, you can set Java breakpoints in the source as well. If you are debugging an ADF application, you should check to see whether ADF declarative breakpoints can be used

instead of Java code breakpoints. For more information, see [Section 29.6, "Setting ADF Declarative Breakpoints"](#).

Before you attempt to use breakpoints, you should try to run the application and look for missing or incomplete data, actions and methods that are ignored or incorrectly executed, or other unexpected results. If you did not find the problem, create a debugging configuration that will enable the ADF Log and send Oracle ADF messages to the Log window. For more information, see [Section 29.4.2, "How to Create an Oracle ADF Debugging Configuration"](#).

To set Java breakpoints to debug an application:

1. Choose **Navigate > Go to Java Class** (or press **Ctrl+Minus**) and use the dialog to locate the Oracle ADF class that represents the entry point for the processing failure.

Note: JDeveloper will locate the class from the user interface project with current focus in the Application Navigator. If your workspace contains more than one user interface project, be sure that the one with the current focus is the one you want to debug.

2. Open the class file in the Java editor and find the Oracle ADF method call that will enable you to step into the statements of the method.
3. Set a breakpoint on the desired method and run the debugger.
4. When the application stops on the breakpoint, use the Data window to examine the local variables and arguments of the current context.

Tip: If you are using the **Go to source** context menu command in the Data, Watches, or Smart Data window, you can go back to the execution point by using the back button. You can also access the back button through the **Navigate** menu.

Once you have set breakpoints to pause the application at key points, you can proceed to view data in the Data window. To effectively debug your web page's interaction with the Oracle ADF Model layer, you need to understand:

- The Oracle ADF page lifecycle and the method calls that get invoked
- The local variables and arguments that the Oracle ADF Model layer should contain during the course of application processing

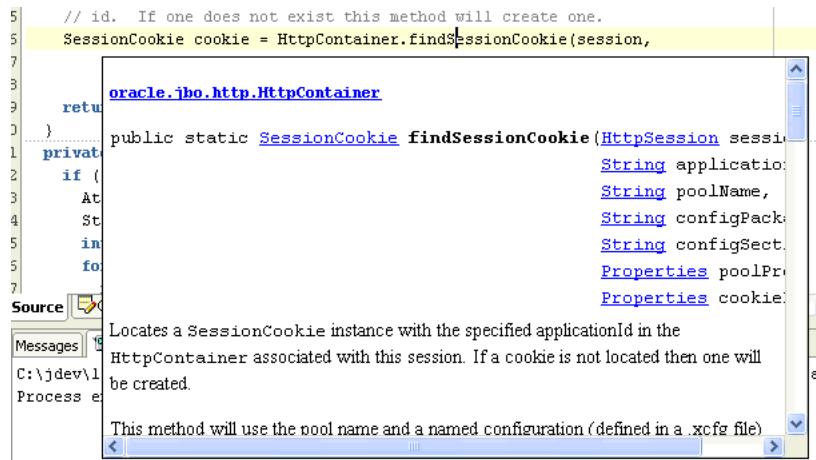
Awareness of Oracle ADF processing will give you the means to selectively set breakpoints, examine the data loaded by the application, and isolate the contributing factors.

Note: JSF web pages may also use backing beans to manage the interaction between the page's components and the data. Debug backing beans by setting breakpoints for them as you would with any other Java class file.

29.7.2 How to Optimize Use of the Source Editor

Once you have added the ADF source library to your project, you have access to the helpful Quick Javadoc feature (**Ctrl+D**) that the source editor makes available.

[Figure 29–22](#) shows Quick Javadoc for a method like `findSessionCookie()`.

Figure 29–22 Using Quick Javadoc on ADF API in the Source Editor

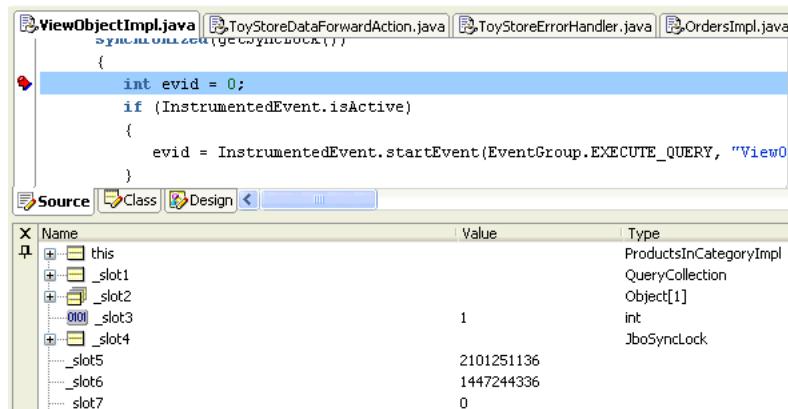
29.7.3 How to Set Breakpoints and Debug Using ADF Source Code

After loading the ADF source code, you can debug any Oracle ADF code for the current project the same way that you do your own Java code. This means that you can press Ctrl+Minus to type in any class name in Oracle ADF, and JDeveloper will open its source file automatically so that you can set breakpoints as desired.

29.7.4 How to Use Debug Libraries for Symbolic Debugging

When debugging Oracle ADF source code, by default you will not see symbol information for parameters or member variables of the currently executing method.

For example, in a debugging session without ADF source code debug libraries, you may see unrecognizable names such as "_slot", as shown in [Figure 29–23](#).

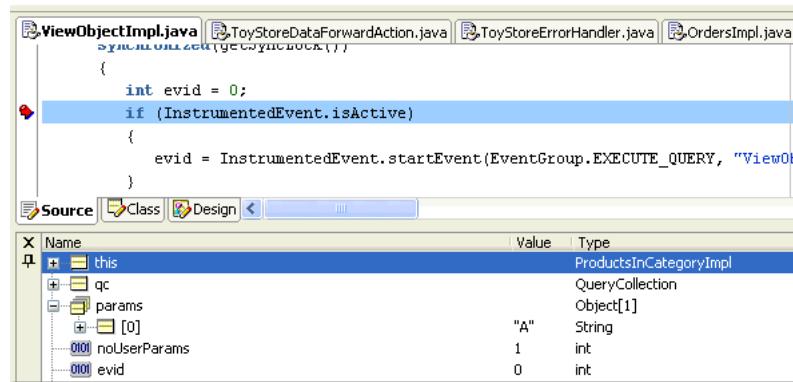
Figure 29–23 Local Symbols Are Hard to Understand Without Debug Libraries

These names are hard to decipher and make debugging more difficult. You can make debugging easier by using the debug versions of the ADF JAR files supplied along with the source while debugging in your development environment.

Note: The supplied debug libraries should not be used in a test or production environment, since they typically have slightly slower runtime performance than the optimized JAR files shipped with JDeveloper.

The debug library JARs are versions of Oracle ADF JARs that have been compiled with additional debug information. When you use these debug JAR files instead of the default optimized JARs, you will see all of the information in the debugger. For example, the variable `evid` is now identified by its name in the debugger, as shown in Figure 29–24.

Figure 29–24 Symbol Information Displayed in Debugger



Before you replace the standard library JAR, make sure that JDeveloper is not running. If it's currently running, exit from the product before proceeding.

To replace the standard library JARs with the debug library JARs:

- With JDeveloper closed, make a backup subdirectory of all existing optimized JAR files in the `./BC4J/lib` directory of your JDeveloper installation. For example, assuming `jdev11` is the JDeveloper home directory:

```
C:\jdev11\BC4J\lib> mkdir backup
C:\jdev11\BC4J\lib> copy *.jar backup
```

- For each ADF Library that you want debug symbols for while debugging, copy the `_g.jar` version of the matching library over the existing, corresponding library in the `C:\jdev11\BC4J\lib` directory.

This is safe to do since you made a backup of the optimized JAR files in the `backup` directory in Step 2.

Since debug libraries typically run a little slower than libraries compiled without debug information, this diagnostic message is to remind you not to use debug libraries for performance timing:

```
*****
*** WARNING: Oracle BC4J debug build executing - do not use for timing ***
*****
```

- To change back to the optimized libraries, simply copy the JAR file(s) in question from the `./BC4J/lib/backup` directory back to the `./BC4J/lib` directory.

29.7.5 How to Use Different Kinds of Breakpoints

You first need to understand the different kinds of breakpoints and where to create them.

To see the debugger Breakpoints window, choose **View > Debugger > Breakpoints** from the main menu or press Ctrl+Shift+R.

You can create a new Java code breakpoint by choosing **Create Breakpoint** from the context menu in the Breakpoints window. The **Breakpoint Type** dropdown list controls what kind of breakpoint you will create, as shown in [Table 29–9](#).

Note: You can use the Edit Breakpoint dialog to edit an ADF declarative breakpoint. However, you cannot edit some of the other information, such as the information in the **Definition** tab. You can launch the Edit Breakpoint dialog by choosing **Edit** from the context menu in the Breakpoint window. For information about creating ADF declarative breakpoints, see [Section 29.6, "Setting ADF Declarative Breakpoints"](#).

Table 29–9 Different Types of Java Breakpoints

Breakpoint Type	The Breakpoint Occurs Whenever	Usage
Exception	An exception of this class (or a subclass) is thrown.	An Exception breakpoint is useful when you don't know where the exception occurs, but you know what kind of exception it is (for example, <code>java.lang.NullPointerException</code> , <code>java.lang.ArrayIndexOutOfBoundsException</code> , <code>oracle.jbo.JboException</code>). The checkbox options allow you to control whether to break on caught or uncaught exceptions of this class. The Browse button helps you find the fully qualified class name of the exception. The Exception Class combobox remembers the most recently used exception breakpoint classes. Note that this is the default breakpoint type when you create a breakpoint in the breakpoints window.
Source	A particular source line in a particular class in a particular package is run.	You rarely create a source breakpoint in the Create Breakpoint dialog, because it's much easier to create it by first using the Navigate > Go to Class menu (accelerator Ctrl+Shift+Minus), then scrolling to the line number you want — or using Navigate > Go to Line (accelerator Ctrl+G) — and finally clicking in the breakpoint margin at the left of the line you want to break on. This is equivalent to creating a new source breakpoint, but it means you don't have to type in the package, class, and line number by hand.

Table 29–9 (Cont.) Different Types of Java Breakpoints

Breakpoint Type	The Breakpoint Occurs Whenever	Usage
Method	A method in a given class is invoked.	The Method breakpoint is useful for setting breakpoints on a particular method you might have seen in the call stack while debugging a problem. If you have the source, you can set a source breakpoint wherever you want in that class, but this kind of breakpoint lets you stop in the debugger even when you don't have source for a class.
Class	Any method in a given class is invoked.	The Class breakpoint can be used when you might know the class involved in the problem, but not the exact method you want to stop on. This kind of breakpoint does not require source. The Browse button helps you quickly find the fully qualified class name you want to break on.
Watchpoint	A given field is accessed or modified.	The Watchpoint breakpoint can be used to find a problem if the code inside a class modifies a member field directly from several different places (instead of going through setter or getter methods each time). You can pause the debugger when any field is modified. You can create a breakpoint of this type by using the Toggle Watchpoint menu item on the context menu when pointing at a member field in your class's source.

29.7.6 How to Edit Breakpoints for Improved Control

After creating a Java code breakpoint you can edit the breakpoint in the Breakpoints window by right-clicking it and choosing **Edit** in the context menu.

Note: You can use the Edit Breakpoint dialog to edit an ADF declarative breakpoint. However, you cannot edit some of the other information such as the information in the **Definition** tab. You can launch the Edit Breakpoint dialog by choosing **Edit** from the context menu in the Breakpoint window. For information about creating ADF declarative breakpoints, see [Section 29.6, "Setting ADF Declarative Breakpoints"](#).

Some of the features you can use by editing your breakpoint are:

- Associate a logical "breakpoint group" name to group this breakpoint with others of the same group name. Breakpoint groups make it easy to enable/disable an entire set of breakpoints in one operation.
- Associate a debugger action to a breakpoint when the breakpoint is hit. The default action is to stop the debugger so that you can inspect the application states, but you can add a sound alert, write information to a log file, and enable or disable group of breakpoints.
- Associate a conditional expression with the breakpoint so that the debugger stops only when that condition is met. The expressions can be virtually any boolean expression, including:
 - *expr ==value*

- `expr.equals("value")`
- `expr instanceof.fully.qualified.ClassName`

Note: Use the debugger Watches window to evaluate the expression first to make sure it's valid.

29.7.7 How to Filter Your View of Class Members

You can use the debugger to filter the members that are displayed in the debugger window for any class. In the debugger's Data window, selecting any item and choosing **Preferences** from the context menu brings up a dialog that lets you customize which members appear in the debugger and (more importantly sometimes) which members *don't* appear. You can filter by class type to simplify the amount of scrolling you need to do in the debugger Data window. This is especially useful when you might be interested only in a handful of a class's members.

29.7.8 How to Use Common Oracle ADF Breakpoints

If you loaded Oracle ADF source code, you can use the breakpoints listed in [Table 29–10](#) to debug your application.

By looking at the Stack window when you hit these breakpoints, and stepping through the source, you can get a better idea of what's going on.

Table 29–10 Commonly Used ADF Breakpoints

Breakpoint	Breakpoint Type	Usage
<code>oracle.jbo.JboException</code>	Exception	This breakpoint useful for setting a breakpoint on the base class of all ADF Business Components runtime exceptions.
<code>oracle.jbo.DMLEception</code>	Exception	This is the base class for exceptions originating from the database, like a failed DML operation due to an exception raised by a trigger or by a constraint violation.
<code>doIt()</code>	Method	<p>You can also perform the same debugging function by setting an ADF declarative breakpoint on the page definition action binding. See Section 29.6.3, "How to Set Page Definition Action Binding Breakpoints".</p> <p>If you prefer to use this Java breakpoint, you can find it in the <code>JUCtrlActionBinding</code> class (<code>oracle.jbo.uicli.binding</code> package).</p> <p>This is the method that will execute when any ADF action binding is invoked, and you can step into the logic and look at parameters if relevant.</p>

Table 29–10 (Cont.) Commonly Used ADF Breakpoints

Breakpoint	Breakpoint Type	Usage
oracle.jbo.server.ViewObjectImpl.executeQueryForCollection	Method	This is the method that will be called when a view object executes its SQL query.
oracle.jbo.server.ViewRowImpl.setAttributeInternal	Method	This is the method that will be called when any view row attribute is set.
oracle.jbo.server.EntityImpl.setAtributeInternal	Method	You can also perform the same debugging function by setting an ADF declarative breakpoint on the page definition attribute value binding. See Section 29.6.4, "How to Set Page Definition Attribute Value Binding Breakpoints" .
		This is the method that will be called when any entity object attribute is set.

29.8 Regression Testing with JUnit

Testing your business services is an important part of your application development process. By creating a set of JUnit regression tests that exercise the functionality provided by your application module, you can ensure that new features, bug fixes, or refactorings do not destabilize your application. JDeveloper's integrated support for creating JUnit regression tests makes it easy test your application. Its integrated support for running JUnit tests means that any developer on the team can run the test suite with a single mouse click, greatly increasing the chances that every team member can run the tests to verify their own changes to the system. Furthermore, by using JDeveloper's integrated support for creating and running Apache Ant build scripts, you can easily incorporate running the tests into your automated build process as well. You can create a JUnit test for your application module, run it, and integrate the tests into an Ant build script.

JDeveloper provides the ability to generate JUnit test cases, test fixtures, and test suites. You can create test cases to test individual Java files containing single or multiple Java classes. You can create JUnit test fixtures that can be reused by JUnit test cases. You can group all these test cases into a JUnit test suite, which you can run together as a unit.

You can also create a separate project to contain your regression tests or integrate the test files into an existing project.

Creating separate projects for testing has the following advantages:

- The ability to compile the base project without having a dependency on JUnit
- The ability to package the base project for deployment without having to exclude the test classes.

If you are creating separate projects for JUnit testing, you should create directory structures that mirror the structure of the packages being tested. You may want to name the test classes using a naming convention that can easily identify the package being tested. For example, if you are testing `myClass.java`, you can name the test class `myClassTest.java`.

Although having separate projects has many advantages, in certain cases it may be easier to include the tests within the project. For example, the Fusion Order Demo application has a JUnit regression test suite in the `FODCustomization` workspace `Customization Extension` project.

You can use the Create Test wizards in the context of the project to create a JUnit test case, test fixture, or test suite. However, if you do not want to include these tests as part of the deployment, you may want to separate the tests out in their own project.

Tip: If you don't see the Create Test wizards, use JDeveloper's **Help > Check for Updates** feature to install the JUnit Integration extension before continuing.

Each test case class contains a `setUp()` and `tearDown()` method that JUnit invokes to allow initializing resources required by the test case and to later clean them up. These test case methods invoke the corresponding `setUp()` and `tearDown()` methods to prepare and clean up the test fixture for each test case execution. Any time a test in the test case needs access to the application module, it uses the test fixture's `getApplicationContext()` method. The method returns the same application module instance, saved in a member field of the test fixture class, between the initial call to `setUp()` and the final call to `tearDown()` at the end of the test case.

JDeveloper supports JUnit 4, which allows annotations to be used instead of explicitly having to name the methods `setUp()` and `tearDown()`. These annotations — `@Before`, `@After` — allow you to have multiple setup and teardown methods, including inherited ones if required.

The generated `ExampleModuleConnectFixture` is a JUnit test fixture that encapsulates the details of acquiring and releasing an application. It contains a `setUp()` method that uses the `createRootApplicationModule()` method of the Configuration class to create an instance of an application module. Its `tearDown()` method calls the matching `releaseRootApplicationModule()` method to release the application module instance.

Your own testing methods can use any of the programmatic APIs available in the `oracle.jbo` package to work with the application module and view object instances in its data model. You can also cast the `ApplicationModule` interface to a custom interface to have your tests invoke your custom service methods as part of their job. During each test, you will call one or more `assertXxxx()` methods provided by the JUnit framework to assert what the expected outcome of a particular expression should be. When you run the test suite, if any of the tests in any of the test cases contains assertions that fail, the JUnit Test Runner window displays the failing tests with a red failure icon.

The JUnit test generation wizard generates skeleton test case classes for each view object instance in the data model, each of which contains a single test method named `testAccess()`. This method contains a call to the `assertNotNull()` method to test that the view object instance exists.

```
// In ViewInstanceNameTest.java test case class
public void testSomeMeaningfulName() {
    // test assertions here
}
```

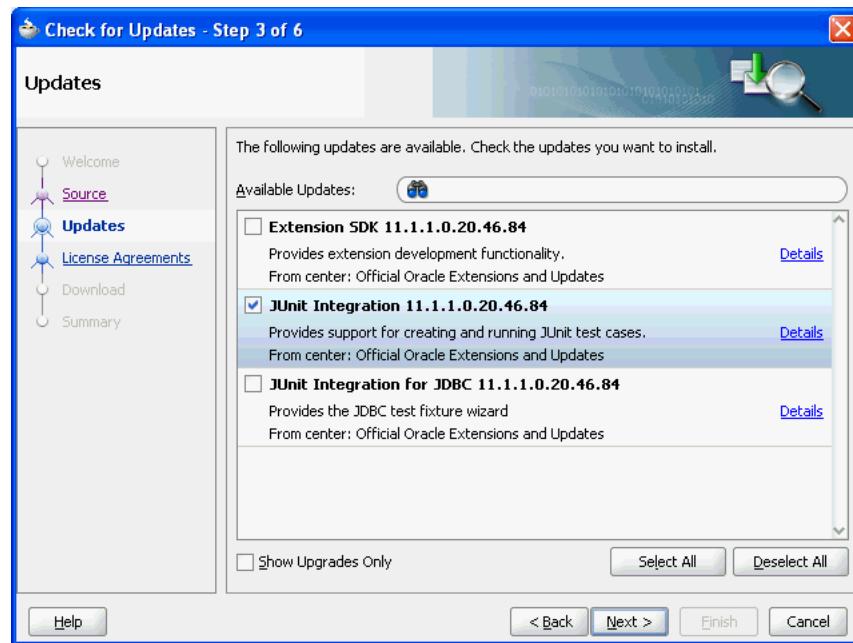
Each generated test case can contain one or more test methods that the JUnit framework will execute as part of executing that test case. You can add a test to the test case simply by creating a public void method in the class whose name begins with the prefix `test` or use the annotation `@Test`.

29.8.1 How to Obtain the JUnit Extension

JUnit must be loaded as an extension to JDeveloper before it becomes available and appears in the menu system.

To load the JUnit extension:

1. From the main menu, choose **Help > Check for Updates**.
2. In the Source page of the Check for Updates dialog, select **Search Update Centers** and **Official Oracle Extensions and Updates** and click **Next**.
3. In the Updates page, select **JUnit Integration** and click **Next**, as shown in [Figure 29–25](#).

Figure 29–25 Check for Updates Dialog for Adding JUnit Extension

4. On the License Agreements page, click **I Accept** and click **Finish**.

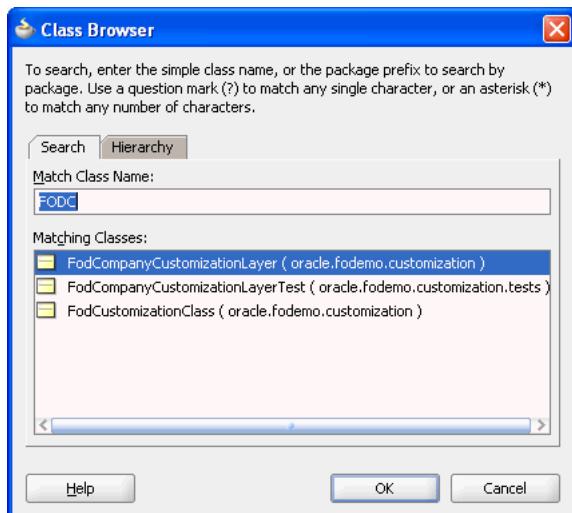
29.8.2 How to Create a JUnit Test Case

Before you create a JUnit test case, you must have created a project that is to be tested.

To generate a JUnit test case:

1. In the Application Navigator, select the project you want to generate a test case for, right-click and select **New**.
2. In the New Gallery, expand **General**, select **Unit Tests(JUnit)** and then **Test Case**, and click **OK**.
3. In the Select the Class to Test page of the Create Test Case dialog, enter the class under test or click **Browse**.
4. In the Class Browser dialog, locate the class you want to test or enter the beginning letters in the **Match Class Name** field. The **Match Class** list will be filtered for easier identification.

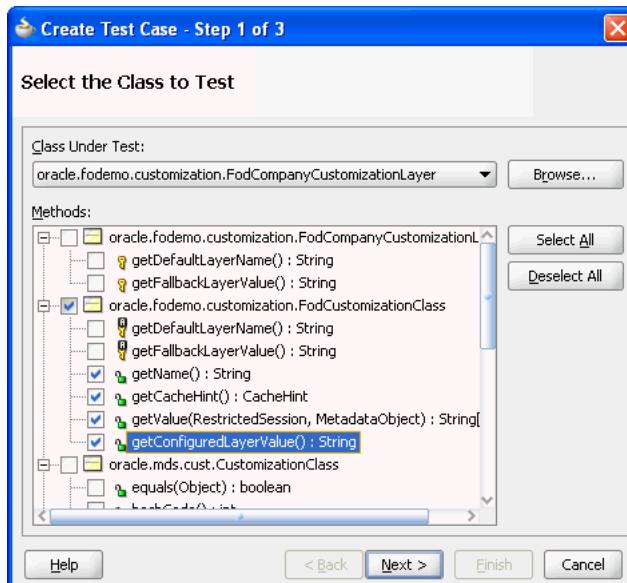
For example, entering FOD filters the list down to three items, as shown in [Figure 29–26](#).

Figure 29–26 Class Browser for Selecting Class Files to Test

Select the class and click **OK** to close the dialog. Click **Next**.

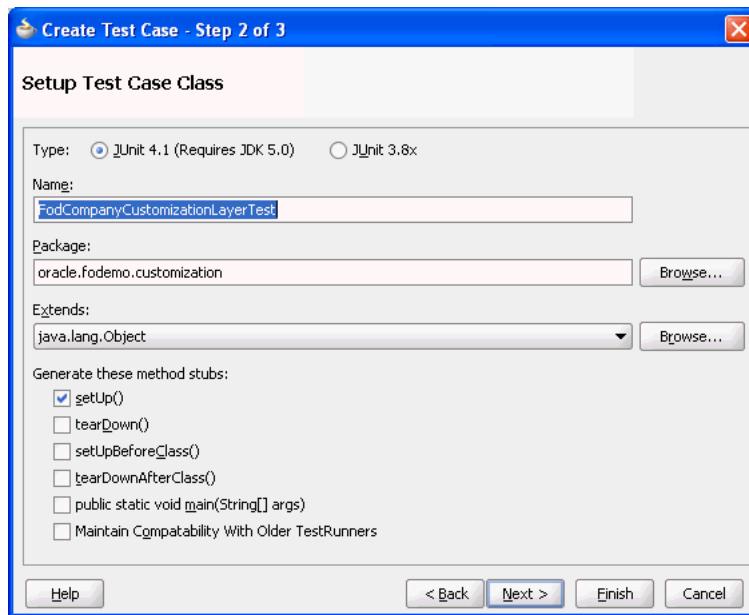
5. Select the individual methods you want to test, and click **Next**.

For example, in [Figure 29–27](#), the four methods that are checked are to be tested.

Figure 29–27 Create Test Case Dialog for Selecting Methods to Test

6. In the Setup Test Case Class page, enter the name of the test case, the package, and the class it extends and select the list of built-in functions JUnit will create stubs for. Click **Next**.

For example, in [Figure 29–28](#), JUnit will create a stub for the `setUp()` method for the `FodCompanyCustomizationLayerTest` test case in the `oracle.fodemo.customization` package.

Figure 29–28 Create Test Case Dialog for Setting up Class to Test

7. In the Select Test Fixtures page, select any test fixtures you want to add to the test case or click **Browse**.
8. Make sure that all the test fixtures you want to add to the test case are selected in the list and click **Finish**.

29.8.3 How to Create a JUnit Test Fixture

You should create a JUnit test fixture if you require more than one test for a class or method. A JUnit text fixture allows you to avoid duplicating test code that is needed to initialize testing.

To generate a JUnit test fixture:

1. In the Application Navigator, select the project you want to generate a test fixture for, right-click and select **New**.
2. In the New Gallery, expand **General**, select **Unit Tests(JUnit)** and then **Test Fixture**, and click **OK**.
3. Select **Test Fixture** and click **OK**.
4. In the Create Test Fixture dialog, enter the name of the test fixture, the package, and any class it extends.
5. Click **OK**.

29.8.4 How to Create a JUnit Test Suite

Before you create a JUnit test suite, you should have already created JUnit test cases that can be added to the test suite.

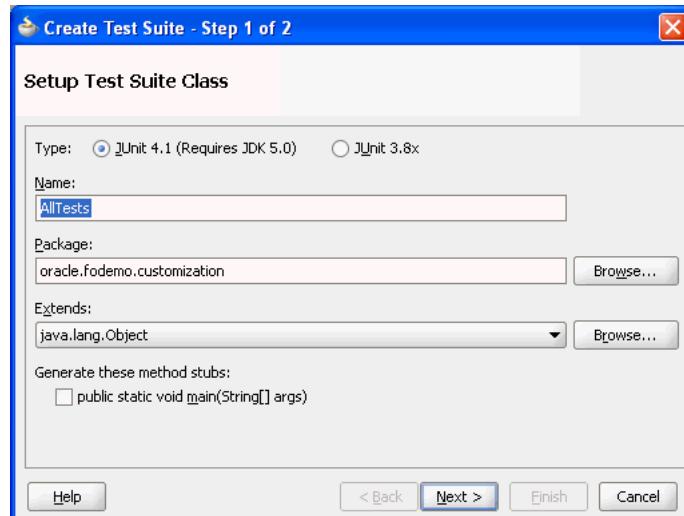
To generate a JUnit test suite:

1. In the Application Navigator, select the project you want to generate a test fixture for, right-click and select **New**.

2. In the New Gallery, expand **General**, select **Unit Tests(JUnit)** and then **Test Suite**, and click **OK**.
3. In the Setup Test Suite Class page of the Create Test Suite dialog, enter the name of the test suite, the package, and the class it extends. Click **Next**.

For example, in Figure 29–29, an `AllTests` test suite is created that extends the `java.lang.Object` class.

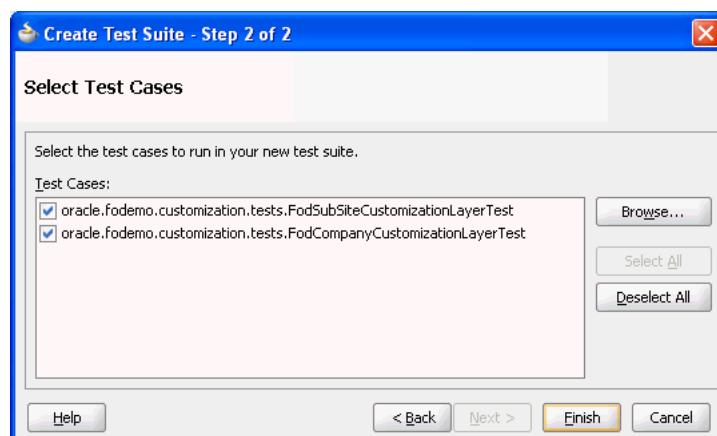
Figure 29–29 Create Test Suite Dialog



4. In the Select Test Cases page of the Create Test Suite dialog, check that all the test cases you want included in the test suite have been selected. The test cases you have created will populate the list. Deselect any test cases that you do not want included. Click **Finish**.

For example, in Figure 29–30, both test cases are selected to be in the test suite.

Figure 29–30 Selecting Test Cases for a Test Suite



29.8.5 How to Run a JUnit Test Suite as Part of an Ant Build Script

Apache Ant is a popular, cross-platform build utility for which JDeveloper offers design time support. You can incorporate the automatic execution of JUnit tests and test output report generation by using Ant's built-in `junit` and `junitreport` tasks.

Example 29–3 shows a task called `tests` from the `FODCustomizations` Ant `build.xml` file in the `CustomizationExtension` project. It depends on the `build` and `buildTests` targets that Ant ensures have been executed before running the `tests` target.

Example 29–3 Ant Build Target Runs JUnit Test Suite

```
<target name="testCustomizations" depends="compileExtensionClasses">
  <junit printsummary="yes" haltonfailure="yes">
    <classpath refid="customization.classpath">
      <pathelement location="${customization.build.dir}" />
    </classpath>
    <formatter type="plain"/>
    <test name="oracle.fodemo.customization.tests.AllTests"/>
  </junit>
</target>
```

The `junit` tag contains a nested `test` tag that identifies the test suite class to execute and specifies a directory in which to report the results. The `junitreport` tag allows you to format the test results into a collection of HTML pages that resemble the format of Javadoc.

To try running the JUnit test from Ant, select the `build.xml` file in the Application Navigator, and choose **Run Ant Target** > *tests* from the context menu.

30

Refactoring a Fusion Web Application

This chapter describes considerations for renaming, moving, and deleting files, configuration files, objects, attributes, and elements. In most cases, JDeveloper automatically performs refactoring. However, you may need to complete some manual steps to refactor.

This chapter includes the following sections:

- [Section 30.1, "Introduction to Refactoring a Fusion Web Application"](#)
- [Section 30.2, "Renaming Files"](#)
- [Section 30.3, "Moving JSF Pages"](#)
- [Section 30.4, "Refactoring pagedef.xml Bindings Objects"](#)
- [Section 30.5, "Refactoring ADF Business Components"](#)
- [Section 30.6, "Refactoring Attributes"](#)
- [Section 30.7, "Refactoring Named Elements"](#)
- [Section 30.8, "Refactoring ADF Task Flows"](#)
- [Section 30.9, "Refactoring the DataBindings.cpx File"](#)
- [Section 30.10, "Refactoring Across Abstraction Layers"](#)
- [Section 30.11, "Refactoring Limitations"](#)
- [Section 30.12, "Refactoring the .jpx Project File"](#)

30.1 Introduction to Refactoring a Fusion Web Application

JDeveloper provides refactoring options to rename, move, and delete the ADF Business Components objects, attributes, and named elements that your application uses. These refactoring options synchronize your changes with other parts of the application that are dependent on the changes. For example, renaming an ADF Business Components object such as a view using the **Rename** option renames any references to it in other XML source files.

30.2 Renaming Files

You can rename files such as configuration files using the following methods:

- In the Application Navigator, select the file and choose **File > Rename** from the main menu.
- In the Application Navigator, right-click the file and choose **Rename**.

- In the source code editor, select a class name, right-click it, and choose **Rename**.

30.3 Moving JSF Pages

In the Application Navigator, you can right-click a JSF page and choose **Refactor > Move to Package** to move the page to another package.

Moving the JSF page to another package updates:

- `faces-config.xml` files that reference the page and its package
- ADF task flows containing views associated with the page
- `DataBindings.cpx` mappings to the page

30.4 Refactoring pagedef.xml Bindings Objects

The `pagedef.xml` binding objects that you can refactor include bindings and executables. For more information, see [Section 11.6, "Working with Page Definition Files"](#).

To refactor `pagedef.xml` binding objects:

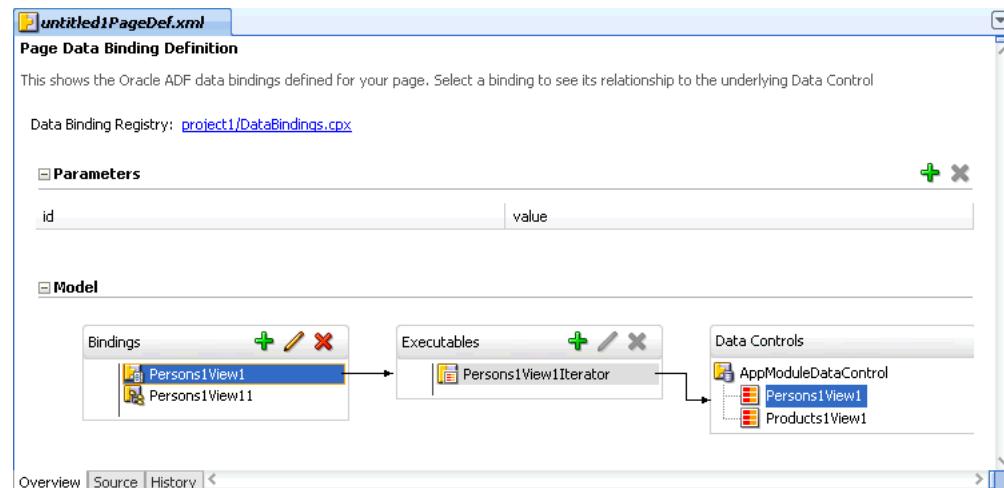
- In the Application Navigator, select the page node on which you have added a bound object such as an ADF Form or selection list.
- Right-click the page node and choose **Go to Page Definition**.

If the page does not already have a page definition, the Create Page Definition dialog displays. Click **OK** to create a page definition for the page.

- In the overview editor, expand the **Model** section.

Data bindings such as list bindings and iterator bindings defined for the page display under **Bindings** and **Executables**, as shown in [Figure 30–1](#).

Figure 30–1 Page Data Binding Definition Overview Tab



- Right-click a data binding or executable, choose **Refactor** and a refactoring option such as **Rename** or **Delete Safely**.
- To display the usages between bindings, executables, and data controls, right-click a binding or executable and choose **Find Usages**.

30.5 Refactoring ADF Business Components

Oracle ADF Business Components includes objects such as views and entities. [Table 30–1](#) shows support for refactoring ADF Business Components.

Table 30–1 Refactoring ADF Business Components

Action	Result
Move	Moves the object to a different package or directory and updates all references.
Delete	JDeveloper shows all dependencies on the object and permits a forced delete. The application may not work at this point. You may need to resolve broken references.
Rename	ADF Business Components objects are defined by an XML file. The XML file has a file name identical to the object name. For example, the name of the XML file for a view object named Persons1View is Persons1View.xml. Renaming results in changing the Name attribute, renaming the XML file, and updating all references. For example, the name of an entity (Customer) is stored as an attribute in the XML file (name=Customer). The XML file has the same name the entity name (Customer.xml).

To refactor ADF Business Components objects:

1. In the Application Navigator, expand the **Projects** node containing the object you want to refactor.
2. Within the project, expand the **Application Sources** node and then the package containing the object you want to refactor.
3. Right-click the object and choose **Refactor > Rename** or **Refactor > Move**.
4. To delete the object, choose **Delete Safely**.

If the object is used elsewhere in the application, a dialog displays with the following options:

- **Ignore:** Unresolved usages will remain in the code as undefined references.
- **View Usages:** Display a preview of the usages of the element in the Compiler log. You can use the log to inspect and resolve the remaining usages.

30.6 Refactoring Attributes

[Table 30–1](#) shows support for refactoring object attributes.

Table 30–2 Refactoring Attributes

Action	Result
Move	Not supported.
Delete	JDeveloper shows all dependencies on the object and permits a forced delete. The application may not work at this point. You may need to resolve broken references.

Table 30–2 (Cont.) Refactoring Attributes

Action	Result
Rename	<p>Attributes share data elements represented in entity and view objects (see Section 4.1, "Introduction to Entity Objects" for more information). References to the attribute are updated when you rename the attribute. Renaming results in changing the Name attribute, renaming the XML file, and updating all references.</p> <p>Renaming an attribute does not change the data it represents, nor does it rename the underlying table column.</p>

To refactor attributes:

1. In the Application Navigator, expand the **Projects** node containing the object you want to refactor.
2. Within the project, expand the **Application Sources** node and then the package containing the object you want to refactor.
3. Double-click the object.
4. In the overview editor, select **Attributes**.
5. In the **Name** column, select an attribute.
6. Right-click and choose **Rename** or **Delete Safely**.
7. To delete the attribute, choose **Delete Safely**.

If the attribute is used elsewhere in the application, a dialog displays with the following options:

- **Ignore:** Unresolved usages will remain in the code as undefined references.
- **View Usages:** Display a preview of the usages of the attribute in the Compiler log. You can use the log to inspect and resolve the remaining usages.

30.7 Refactoring Named Elements

[Table 30–3](#) shows support for refactoring named elements in an XML schema. Named elements are any elements in the XML schema that can be referenced by a **Name** attribute. A named element is not an object or an attribute.

Table 30–3 Refactoring Named Elements

Action	Result
Move	Not supported.
Delete	Not supported.
Rename	One exception to the definition of named elements is the design time element <code>Attr</code> , which does have a Name attribute. <code>Attr</code> is a name-value pair, is not accessible from the main editor, and should not be renamed.

To refactor named elements:

1. In the Application Navigator, expand the **Projects** node containing the object you want to refactor.
2. Within the project, expand the **Application Sources** node and then the package containing the object you want to refactor.
3. Double-click the object.

4. In the overview editor, click the **Source** tab.
5. Scroll down to a named element.

Named elements are indicated by `Name= "<element>"` in the source code, for example:

```
<Key Name="PersonsAffContactChk">
```

A named element is not an object or attribute.

6. To rename the element, right-click the element and choose **Rename**.
7. To delete the element, choose **Delete Safely**.

If the element is used elsewhere in the application, a dialog displays with the following options:

- **Ignore**: Unresolved usages will remain in the code as undefined references.
- **View Usages**: Display a preview of the usages of the element in the Compiler log. You can use the log to inspect and resolve the remaining usages.

30.8 Refactoring ADF Task Flows

For more information, see [Section 13.7, "Refactoring to Create New ADF Task Flows and Templates"](#).

30.9 Refactoring the DataBindings.cpx File

The DataBindings.cpx file defines the Oracle ADF binding context for the entire application and provides the metadata from which the Oracle ADF binding objects are created at runtime (see [Section A.6.1, "DataBindings.cpx Syntax"](#) for more information). This file is a registry used to quickly find all .cpx, .dcx, .jpx, and .xcfg files, which are themselves registries of metadata.

If you rename the DataBindings.cpx file to a new name, such as DataBindingsNew.cpx, the change is added to the adfm.xml file.

[Example 30–1](#) shows the contents of the adfm.xml file after DataBindings.cpx is refactored to DataBindingsNew.cpx.

Example 30–1 Renamed DataBindings.cpx file in adfm.xml

```
<?xml version="1.0" encoding="UTF-8" ?>
<MetadataDirectory xmlns="http://xmlns.oracle.com/adfm/metainf"
version="11.1.1.0.0">
<DataBindingRegistry path="adf/sample/view/DataBindingsNew.cpx"/>
</MetadataDirectory>
```

After this change, the application won't run because the data control cannot be found. [Example 30–2](#) shows the old ID in the DataBindingsNew.cpx file. To enable the application to access the correct bindings file, update the ID value in the DataBindingsNew.cpx file to the new file.

Example 30–2 DataBindingsNew.cpx ID

```
<Application xmlns="http://xmlns.oracle.com/adfm/application"
version="11.1.1.49.28" id="DataBinding" SeparateXMLFiles="false"
Package="adf.sample.view" ClientType="Generic">
```

This should be changed to an ID similar to the one shown in [Example 30–3](#).

Example 30–3 Updated DatabindingNew.cpx ID

```
<Application xmlns="http://xmlns.oracle.com/adfm/application"
version="11.1.1.49.28" id="DataBindingNew" SeparateXMLFiles="false"
Package="adf.sample.view" ClientType="Generic">
```

30.10 Refactoring Across Abstraction Layers

Refactoring does not cross abstraction layers. For example, when a view object is created based on the entity `Dept`, it is named `DeptView` by default. Renaming `Dept` updates the entity usage in `DeptView`, but does not change the name of the view object.

30.11 Refactoring Limitations

[Table 30–4](#) summarizes the limitations of JDeveloper's refactoring support.

Table 30–4 Refactoring Limitations

Area	Limitation
Database	When a database artifact used in an ADF Business Components object is renamed, the object needs to be updated. This type of refactoring is currently not supported.
Java literal references	The Java code generated by ADF Business Components has literal references to the XML metadata. These literal references are updated during refactoring operations. Generated (type safe) methods are also updated. However, if the application directly refers to the metadata, these references are not updated.
Domain	If a domain needs to be renamed or moved, you must create the new domain, then change the type of existing domain usages. For example, you might rename a domain called <code>EmployeeID</code> to <code>EmployeeNumber</code> . In addition, the entity <code>Emp</code> has an attribute called <code>Empno</code> that is of type <code>EmployeeID</code> . After creating the new domain <code>EmployeeNumber</code> , go to the attributes page of the entity editor, right-click <code>Empno</code> and choose Change Type . This switches <code>Empno</code> from <code>EmployeeID</code> to <code>EmployeeNumber</code> .
Security	Entity object and attribute security policies in the policy store may reference the name of an entity object or attribute. These policy definitions are not updated in response to the refactoring of an entity object or attribute.
Resource Bundles	Entity object definitions can reference a resource in one or more arbitrary resource bundle (.properties) files that you create. You can use this file to define labels for the attributes of entity objects. However, if you rename the .properties file you created, JDeveloper will not update the entity object definitions to reflect the new file name. As an alternative to resource bundle files that you create, you can specify a project setting to generate a single, default resource bundle file for the ADF Business Components project. In this case, JDeveloper will not allow you to rename this generated file. However, if you attempt to change the project-level default resource bundle file, JDeveloper will warn you about the change. The ADF Business Components project will honor the new project-level setting for any objects that have not yet been linked to the default resource bundle file; all existing Business Components that have already been linked to the original default file will continue to use it instead.

Table 30–4 (Cont.) Refactoring Limitations

Area	Limitation
.jpx project file	<p>Renaming the ADF Business Components (.jpx) project file is not supported.</p> <p>In previous versions of JDeveloper, ADF Business Components .jpx files were created only in the root package of the <code>src</code> directory of a project and were named with the same base name as the project. The ADF Business Components objects (entities, views, and application modules) were all created in the model package (for example, <code>/model/AppModule.xml</code>), but the <code>/Model.jpx</code> is not.</p> <p>This may cause a reusability problem when attempting to package them in ADF JAR files for use on the class path. There may be name conflicts because several projects are named Model.</p>

30.12 Refactoring the .jpx Project File

JDeveloper supports refactoring ADF Business Components.

To manually move the .jpx project file:

1. Move the .jpx file to the new source tree location.

For example, you can move the `Model.jpx` file from `src/Model.jpx` to `src/packagenames/go/here/Model.jpx`.

2. Change the .jpx file contents. The `PackageName` attribute of the root element `JboProject` needs to have the correct value.

For example, you can specify `PackageName="packagenames.go.here"`.

3. Change the `jbo.project` attributes in all `common/bc4j.xcfg` files that contain elements referred to in the .jpx file to include the new package name.

For example:

```
< AppModuleConfig name="ScottDeptAMLocal"
  ApplicationName="packagenames.go.here.ScottDeptAM"
  DeployPlatform="LOCAL" JDBCName="scottdb"
  jbo.project="packagenames.go.here.Model">
```

4. Change the JDeveloper project file (the .jpr file) contents:

- Set the new .jpx package location. If you later change the default package in the project properties, you will again raise a `NotFound` error for the .jpx file.

For example:

```
<value n="defaultPackage" v="packagenames.go.here" />
```

- Fix any `ownerURL` elements in the `ownerMap` that contain references to the old location of the .jpx file.

For example:

```
<url n="ownerURL"
  path="src/packagenames/go/here/Model.jpx" />
```


31

Reusing Application Components

This chapter describes how to package certain ADF components into the ADF Library for reuse in applications. Reusable ADF components are application modules, Business Components, data controls, task flows, page templates, and declarative components.

This chapter includes the following sections:

- [Section 31.1, "Introduction to Reusable Components"](#)
- [Section 31.2, "Packaging a Reusable ADF Component into an ADF Library"](#)
- [Section 31.3, "Adding ADF Library Components into Projects"](#)
- [Section 31.4, "Removing an ADF Library JAR from a Project"](#)

31.1 Introduction to Reusable Components

In the course of application development, certain components will often be used more than once. Whether the reuse happens within the same application, or across different applications, it is often advantageous to package these reusable components into a library that can be shared between different developers, across different teams, and even across departments within an organization.

In the world of Java object-oriented programming, reusing classes and objects is just standard procedure. With the introduction of the model-view-controller (MVC) architecture, applications can be further modularized into separate model, view, and controller layers. By separating the data (model and business services layers) from the presentation (view and controller layers), changes to any one layer do not affect the integrity of the other layers. You can change business logic without having to change the UI, or redesign the web pages or front end without having to recode domain logic.

Oracle ADF and JDeveloper support the MVC design pattern. When you create an application in JDeveloper, you can choose many application templates that automatically set up model and view-controller projects. Because the different MVC layers are decoupled from each other, development can proceed on different projects in parallel and with a certain amount of independence.

ADF Library further extends this modularity of design by providing a convenient and practical way to create, deploy, and reuse high-level components. When you first design your application, you design it with component reusability in mind. If you created components that can be reused, you can package them into JAR files and add them to a reusable component repository. If you need a component, you may look into the repository for those components and then add them into your project or application.

For example, you can create an application module for a domain and package it to be used as the data model project in several different applications. Or, if your application will be consuming components, you may be able to load a page template component from a repository of ADF Library JARs to create common look and feel pages. Then you can put your page flow together by stringing together several task flow components pulled from the library.

[Table 31–1](#) lists the reusable components supported by ADF.

Table 31–1 Oracle ADF Reusable Components

Reusable Component	Description
Data control	Any data control can be packaged into an ADF Library JAR. Some of the data controls supported by Oracle ADF include application modules, Enterprise JavaBeans, web services, URL services, JavaBeans, and placeholder data controls.
Application module	If you are using Business Components and you generate an application module, an associated application module data control is also generated. When you package an application module data control, you also package up the Business Components associated with that application module. The relevant entity objects, view objects, and associations will be a part of the ADF Library JAR and available for reuse.
Business components	Business Components are the entity objects, view objects, and associations used in the business layer. You can package Business Components by themselves or together with an application module.
Task flows and taskflow templates	<p>Task flows can be packaged into an ADF Library JAR for reuse.</p> <p>ADF bounded task flows built using pages can be dropped onto pages. The drop will create a link to call the bounded task flow. A task flow call activity and control flow will automatically be added to the task flow, with the view activity referencing the page. If there is more than one existing task flow with a view activity referencing the page, it will prompt for the application developer to select the one to automatically add a task flow call activity and control flow. If you drop a bounded task flow that uses page fragments, JDeveloper adds a region to the page and binds it to the dropped task flow.</p> <p>If an ADF task flow template was created in the same project as the task flow, the ADF task flow template will be included in the ADF Library JAR and will be reusable.</p>
Page templates	You can package a page template and its artifacts into an ADF Library JAR. If the template uses image files and they are included in a directory within your project, these files will also be available for the template during reuse.
Declarative components	You can create declarative components and package them for reuse. The tag libraries associated with the component will be included and loaded into the consuming project.

You can also package up projects that have several different reusable components if you expect that more than one component will be consumed. For example, you can create a project that has both an application module and a bounded task flow. When this ADF Library JAR file is consumed, the application will have both the application module and the task flow available for use. You can package multiple components into one JAR file, or you can package a single component into a JAR file. Oracle ADF and JDeveloper give you the option and flexibility to create reusable components that best suit you and your organization.

You create a reusable component by using JDeveloper to package and deploy the project that contains the components into a ADF Library JAR file. You use the components by adding that JAR to the consuming project. At design time, the JAR is added to the consuming project's class path and so is available for reuse. At runtime, the reused component runs from the JAR file by reference. For the procedure to add the JAR manually, see [Section 31.3.2, "How to Add an ADF Library JAR into a Project Manually"](#). For the procedure to add the JAR using the JDeveloper Resource Catalog,

see [Section 31.3.1, "How to Add an ADF Library JAR into a Project using the Resource Palette"](#).

Before you proceed to create reusable components, you should review the guidelines for creating reusable components.

31.1.1 Creating Reusable Components

Creating and consuming reusable components should be included in the early design and architectural phases of software projects. You and your development team should consider which components are candidates for reuse, not only in the current applications but also for future applications and including those applications being developed in other departments.

You and your team should decide on the type of repository needed to store the library JARs, where to store them and how to access them. You should consider how to organize and group the library JARs in a structure that fits your organizational needs. You should also consider creating standardized naming conventions so that both creators and consumers of ADF Library JARs can readily identify the component functionality.

Tip: If, in the midst of development, you and your team find a module that would be a good candidate for reuse, you can use the extensive refactoring capabilities of JDeveloper to help eliminate possible naming conflicts and adhere to reusable component naming conventions.

31.1.1.1 Naming Conventions

When you create reusable components, you should try to create unique and relevant names for the application, project, application module, task flow, connection, or any other file or component. Do not accept the JDeveloper wizard default names such as Application, Project, ViewController, AppModule, `task-flow-defintion.xml`, or Connection. You want to try to have unique names to avoid naming conflicts with other projects, components, or connections in the application. Naming conflicts could arise from components created in the consuming application and those loaded from other JAR files. [Table 31–2](#) lists the objects that you may be required to rename.

Table 31–2 Example Unique and Relevant Names for Reusable Components

Type	JDeveloper Default	Example
Application	Application	FusionOrderDemo
Project	Model	OrderBookingService
	ViewController	StoreFrontUI
	Project	
Package	Various possibilities. For more information, see Section 31.1.1.1 .	oracle.foddemo.storefront
Application module	AppModule	StoreServiceAMDataControl
Connection	Connection1	oracle_apps_foddb
Task flow	<code>task-flow-defintion.xml</code>	<code>checkout-task-flow.xml</code>

Table 31–2 (Cont.) Example Unique and Relevant Names for Reusable Components

Type	JDeveloper Default	Example
Page template	templateDef.jspx	StoreFrontTemplate.jspx
Declarative Component	componentDef componentDef.jspx	FODsuperwidgetDef FODsuperwidgetDef.jspx
ADF Library JAR file	adflib<string or 3-digit random number>N	adflibStoreFrontService

For more information, see
[Section 31.1.2.](#)

31.1.1.1 Naming Considerations for Packages

Be aware that some components use the default package name of the project without allowing the name to be explicitly set. In this situation, you must take extra care to avoid package name collisions. You can set the package name in the application creation wizard and you should check the names in the Project Properties dialog afterwards. If you don't set the package name, it will default to a variant of the project name, typically with the first letter being lowercase. For example, a project with the name `Project1` will have a default package name of `project1`. You should manually change the package name to a more unique name before you proceed to build the project.

Note: The basic package naming requirement is that ADF metadata registries (`.dcx`, `.cpk`, and so on) are generated based on the project's package name, and you should avoid metadata naming conflicts between projects that will be combined at runtime.

When you are creating a reusable component's web resources files, such as JSPs, HTMLs, and task flows, you should create them in their own relative directories. When the JAR is deployed into another application, there will be less chance for conflict between the reusable component's files and the consuming application's files.

31.1.1.2 Naming Considerations for Connections

Oftentimes, several modules in an application will connect to the same data source. You should standardize the connection name to the same data source to avoid confusion because there is only one namespace for connections across the application. This would require coordination with other developers, component producers, and component consumers. For example, if `module1` and `module2` both have a connection to the same database, the connection name should be standardized to an agreed upon name, such as `orders_db`.

ADF Library JARs (with connections) may be used in different applications with different connection requirements. ADF Library JAR producers should choose connection names that are at least representative of the connection source, if not the actual standardized connection name. Be aware that consumers of the JAR that was created with connections will be required to satisfy the connection requirements when they add the component to the application.

For example, for a database connection, choosing an endpoint host name is usually not appropriate. The most appropriate name is a complete representative for the schema. Acceptable names for connections are `oracle-appsdb` and `oracle-scottdb`. You

should realize that if many reusable components use different names for the same logical connection, then the consumer of the component will have to satisfy each one individually with duplicate information. The consumer will have to supply connection details for several different connection names, when in fact they all refer to the same instance.

31.1.1.3 Naming Considerations for Applications with EJB Projects

If an application has both EJB projects and a web application project with data binding, you should check to see that the EJB component names are not in conflict with any other web application project component names. The EJB project components may have global scope because the project is automatically added to the global class path of all web projects in the application. Web application projects may mistakenly access a component with the same name in the EJB project rather than within its own project.

For example, if both an EJB project and a web-based project have a `test1.jspx` page, when the web-based project is run, it may try to run the EJB project `test1.jspx` page.

At runtime, JDeveloper detects if there are EJB projects and web application projects with data binding in the same application. If there are both types in the application, when the project is run and the server starts up, a warning message will appear in the Log window.

31.1.1.2 The Naming Process for the ADF Library JAR Deployment Profile

Before you package the project, you must create a deployment profile with the name and path for the JAR file. You should choose a name that follows your development team's naming convention and that is descriptive of the function of the component. You should realize that the consuming project may also include other JAR files from other software authors.

For example, if the component is a task flow for self-service paying, you might name it `mycompany.hcm.pay.selfservice.taskflows.jar`. Other examples are `oracle.apps.hcm.pay.model.jar` and `mycompany.hcm.pay.model.overrides.jar`.

If you do not enter a name, JDeveloper will present a default name with the following format:

`adflibidentifiern`

Where *identifier* is created with the following order of precedence:

1. Name of the project, other than Model, ViewController, or Project.
2. Name of the application, other than application.
3. A random three-digit number.

And n is a number that starts with 1 and increments by 1 for each iteration of the profile.

For example, if the project name were `StoreFrontService`, the profile name would be `adflibStoreFrontService1`. If the project name were `Model`, JDeveloper would reject this name and move to the second order of precedence, application name. If the application name were `FusionOrderDemoShared`, then the project name would be `adflibFusionOrderDemoShared1`. If neither the model nor the application name is usable, then a random three-digit number would be used instead, for example, `adflib7491`.

31.1.1.3 Keeping the Relevant Project

When you are creating reusable components, you should eliminate any projects that are not relevant to the reusable component. For example, if you want to create a reusable application module, you would need a data model project, but you would not need a view-controller project. In this instance, if you had created an application with both a data model and a view-controller project, you could delete the view-controller project. Of course, you should rename the default name `Model` to something more relevant, such as `StoreFrontService`. Similarly, if you are creating a reusable task flow that is not databound, you can delete any data model project from the application.

31.1.1.4 Selecting the Relevant Technology Scope

If you know the technology scope of your consuming projects, you can design your component with technologies that will be compatible. For example, if the consuming application uses only standard JSF Faces, then it may not be compatible with a declarative component that is built with ADF Faces.

When you create your application, you can define the technology scope using the Create Application Wizard by selecting from the application template.

After the project has been created, you can define the technology using the Technology Scope page of the Project Properties dialog.

31.1.1.5 Selecting Paths and Folders

If you are using the file system to store your ADF Library JARs, you should select file system locations that can function as repositories. You may want to put groups of JARs into a common directory folder, for example,

`C:\ADF11\jdev\DevTeamADFRepository`. If your team or organization plans to share ADF Library JARs, you should consider setting up network accessible repository folders, directories, or services. The Resource Palette has provisions to connect to different repository sources and make multiple connections. For more information about accessing ADF Library JARs using the Resource Palette, see [Section 31.1.2, "Using the Resource Palette"](#).

31.1.1.6 Including Connections Within Reusable Components

If the project you are packaging into an ADF Library JAR includes a connection, that information can be included in the JAR and may be available to the consuming project. Oracle ADF uses connection architecture which defines a connection as two parts, the connection name and the connection details (or endpoint definition). JDeveloper will present the producer of the JAR the option to package the connection name only or to include connection details with the connection name.

If a connection is present in the project, the packaged ADF Library JAR will contain a `jar-connections.xml` file and a `jar-adf-config.xml` file. They will be added to the META-INF directory. The `jar-connections.xml` file contains the connection name and other relevant connection information. The `jar-adf-config.xml` file stores the information about the credentials used for the connections. If connection credentials were also specified, then a `jar-credential-jazn-data.xml` will also be included for the credential store.

When an ADF Library JAR is being added to a project, JDeveloper checks for conflicts between the application's connections and the JAR's connections. A dialog will be presented to allow you to decide whether to proceed with adding the JAR to the project. Connections defined in the ADF Library JAR may be added to the consuming project, depending on the following conditions:

- If the connection defined in the JAR is fully configured and there are no connection name conflicts with the consuming project, a new connection will be added.
- If the connection defined in the JAR is partially configured, a new connection will be added to the consuming project but it must be configured before use. Connection dialogs may appear to allow the user to enter connection information. This partially configured connection may be indicated by an incomplete icon.
- If the connection defined in the JAR has the same name as the application's connection, it will not be added to the project. The application's existing connection will be used.
- If the connection defined in the JAR has the same name as the application's connection but is of different type, the JAR's connection will not be added to the project. For example, if a database connection in the JAR and a URL connection in the application have the same name, the database connection will not be added to the application.

For instructions on how to add an ADF Library to a project that includes connections, see [Section 31.3.1, "How to Add an ADF Library JAR into a Project using the Resource Palette"](#).

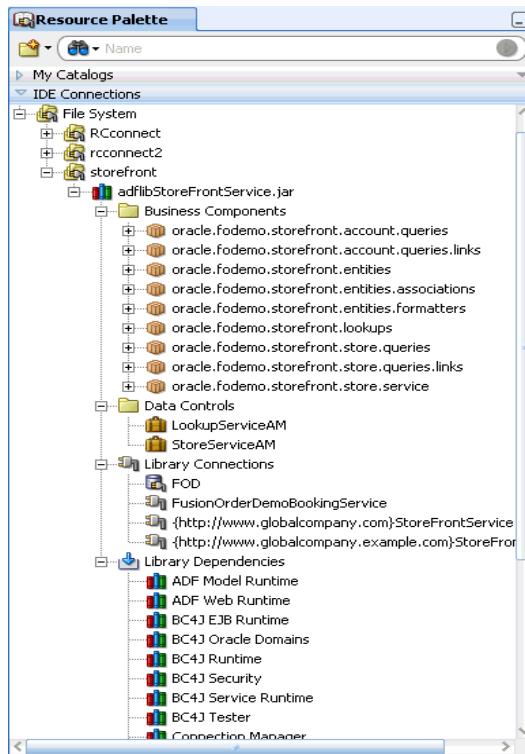
31.1.2 Using the Resource Palette

ADF Library JARs can be packaged, deployed, discovered, and consumed like any other Oracle Library component. Creating an ADF Library JAR is the action of packaging all the artifacts (and additional control files) of a project into a JAR. Consuming a reusable component from an ADF Library JAR is the action of loading that ADF Library JAR into the project's set of libraries.

However, the easiest way to manage and use ADF Library JAR components is by using JDeveloper's Resource Palette. With the Resource Palette, developers who want to consume reusable components can easily find and discover available components and add them to their projects. The Resource Palette provides search and browse functions across different data management systems to locate the component. It provides multiple connections to access different sources. It has a structure tree view for displaying different connections and the ADF Library JAR component types. [Figure 31-1](#) shows the Resource Palette window with three file system connections, RCconnect, rcconnect2, and StoreFront. In this example, StoreFront contains the ADF Library component, adflibStorFrontService.jar.

The Resource Palette tree structure displays each JAR as subcategories. Separate nodes are created for each type of reusable component. For example, application modules are under the Data Controls node, and task flows are under the ADF Task Flows node.

The tree structure for the ADF Library JAR lists any connection information under a Library Connections node and lists all the producing project's extension libraries under the Library Dependencies node.

Figure 31–1 Resource Palette Showing ADF Library Structure

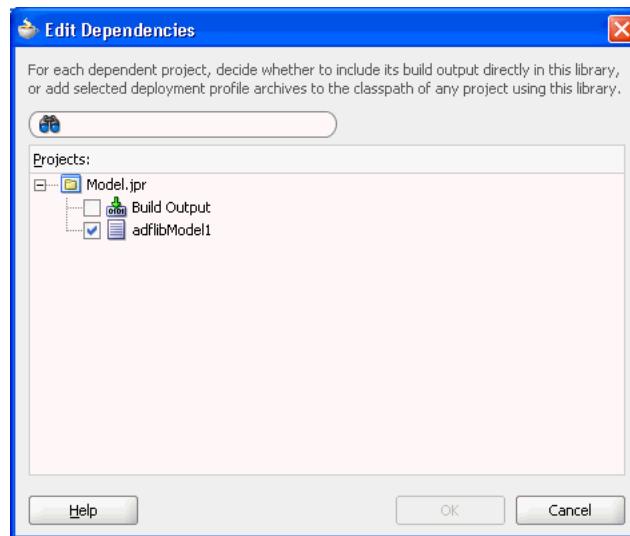
31.1.3 Extension Libraries

An ADF project usually includes a list of extension libraries that it needs to run. These libraries are loaded in the class path of the project. You can view a project's dependent libraries by selecting the **Libraries and Classpath** node of the Project Properties dialog. Some of the libraries that may appear are JSP Runtime, ADF Page Flow Runtime, Connection Manager, and Oracle JDBC.

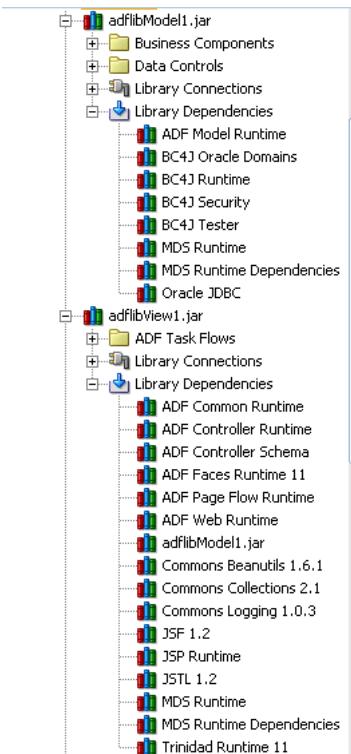
When a project is packaged into an ADF Library JAR, its extension libraries are packaged with it. And when an ADF Library JAR is being consumed by another project, JDeveloper automatically resolves any extension library conflicts between them. During the consuming process, JDeveloper checks to see whether the consuming project already has the extension libraries of the ADF Library JAR in its class path and loads only those libraries that it does not have. For example, if JSP Runtime already exists in the consuming project, it will not be loaded again if the ADF Library JAR also includes it. The consuming project's extension libraries will be a union of its own libraries and the libraries in the ADF Library JAR.

If the project you want to package into an ADF Library has a dependent project, you can include the dependent project's extension libraries directly in the JAR or, if the dependent project has a deployment profile, you can add the dependent project's JAR to the ADF Library. For more information about setting up the deployment process, see [Section 31.2.1, "How to Package a Component into an ADF Library JAR"](#).

For example, project **View** is being packaged into ADF Library **adflibView1.jar**. It has a dependency on the **Model** project. For project **View**, the **Model** project is a dependent project with the deployment profile option (**adflibmodel1**) selected, as shown in [Figure 31–2](#).

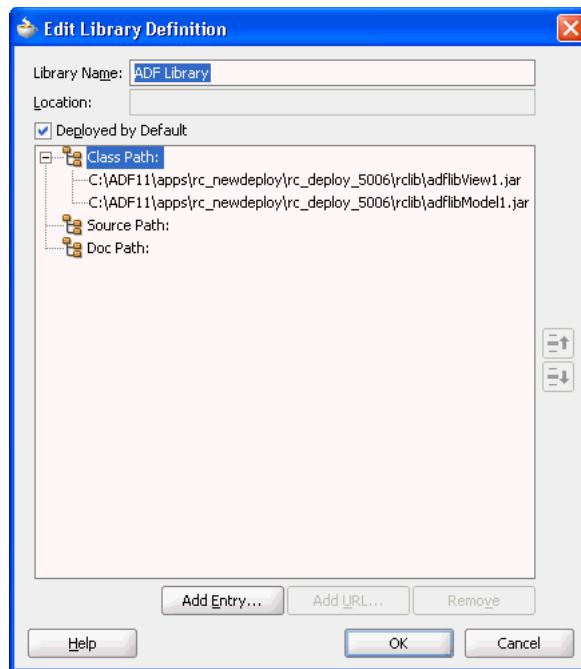
Figure 31–2 Edit Dependencies Dialog with Deployment Profile Option

When the deployment profile is selected, the dependent project's JAR file will be added to the ADF Library JAR. As a result, the extension libraries of the dependent project will also be made available to any consuming project. In [Figure 31–3](#), the ADF Library being packaged, `adflibView1.jar`, includes the dependent `adflibModel1.jar` as listed under the Library Dependencies node.

Figure 31–3 Resource Palette Showing `adflibView1.jar` and `adflibModel1.jar` Extension Libraries

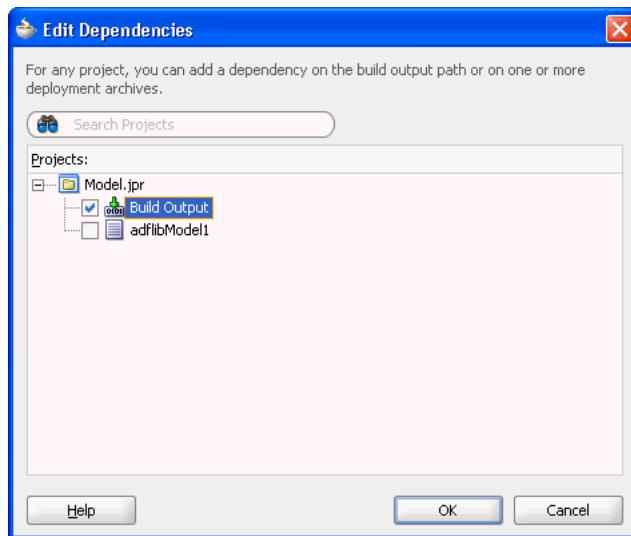
When you select `adflibView1.jar` in the Resource Palette and choose **Add to Project** from the context menu, both `adflibView1.jar` and `adflibModel.jar` will be added to the consuming project's class path.

Figure 31–4 Class path of Consuming Project Showing ADF Library JAR and Dependent Project Library JAR

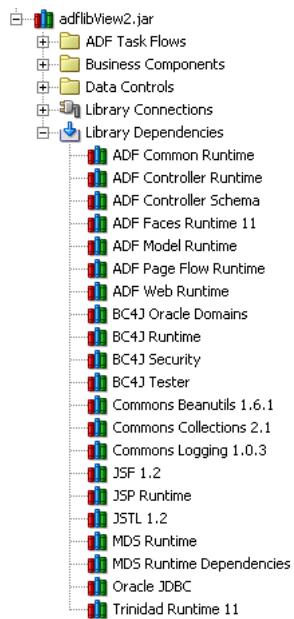


Alternately, you can include the dependent project's artifacts and extension libraries directly into the ADF Library JAR.

For example, project View can also be packaged into ADF Library `adflibView2.jar`. It also has a dependency on the Model project. But in this second deployment profile, the Model project is a dependent project with the **Build Output** option selected (as opposed to the deployment profile (`adflibmodel1`) being selected), as shown in [Figure 31–5](#).

Figure 31–5 Edit Dependencies Dialog Used to Built adflibview2.jar

When **Build Output** is selected, the dependent project's classes and extension libraries will be added directly to the ADF Library JAR. [Figure 31–6](#) shows the ADF Library `adflibView2.jar`, which includes artifacts of the `Model` project and its extension libraries. Note that the extension libraries under the `adflibView2.jar` Library Dependencies node are the same as the combined extension libraries under the `adflibView1.jar` and `adflibModel1.jar` shown in [Figure 31–3](#).

Figure 31–6 Resource Palette Showing adflibView2.jar Extension Libraries

How you decide to package the dependent project depends on how you intend the ADF Library to be used. Including the dependent project as a JAR may be advantageous when the dependent project is itself a reusable component, or when it is a dependent project for several projects that will be packaged into an ADF Library JAR. In the example, the dependent project `Model` may be a dependent project for several view projects. On the other hand, packaging the dependent project as **Build Output** is straightforward and eliminates the need for multiple JARs.

31.2 Packaging a Reusable ADF Component into an ADF Library

Once you have decided that a certain component or components can be reused, create an application and a project to develop that component. Follow the guidelines in [Section 31.1.1, "Creating Reusable Components"](#) to name your application, project, package, and other objects and files. A project corresponds to one ADF Library JAR. If you create multiple projects and want to reuse components from each of the projects, you may need to create an ADF Library JAR for each project. In other situations, you may be able to involve multiple components under one project to create a single ADF Library JAR. For example, you may be able to create Business Components, application modules, task flows, and page templates all under one project and create one ADF Library JAR.

Creating an ADF Library JAR involves compiling the project and validating the components, creating a resource service file, control files, an `adflibREADME.txt`, and adding the relevant project files into a JAR. For more information about the ADF Library JAR, see [Section 31.2.2, "What Happens When You Package a Project to an ADF Library JAR"](#).

If you are packaging a component that itself uses another ADF Library component, the final consuming project must have both ADF Library JARs added to the project. For example, say you created a reusable task flow that contains tables dropped from a data control in another ADF Library JAR. When you add the task flow from an ADF Library JAR into a consuming project, that project will also require the data control ADF Library JAR.

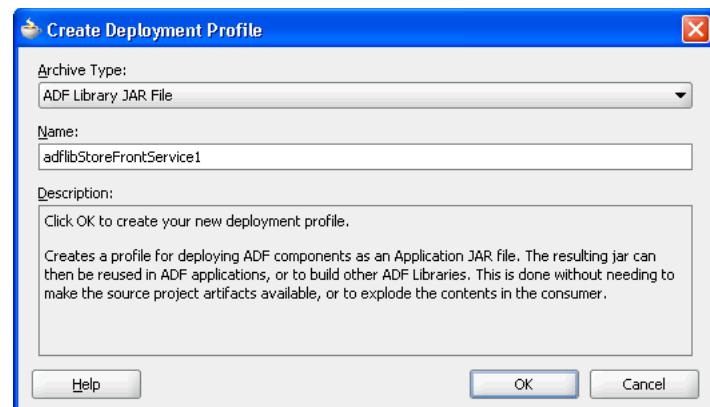
31.2.1 How to Package a Component into an ADF Library JAR

To package up a reusable component, you first create a deployment profile that specifies the archive type, the name of the JAR file, and the directory path where the JAR will be created. Then you deploy the project using the deployment profile.

To package and deploy a project into the ADF Library JAR:

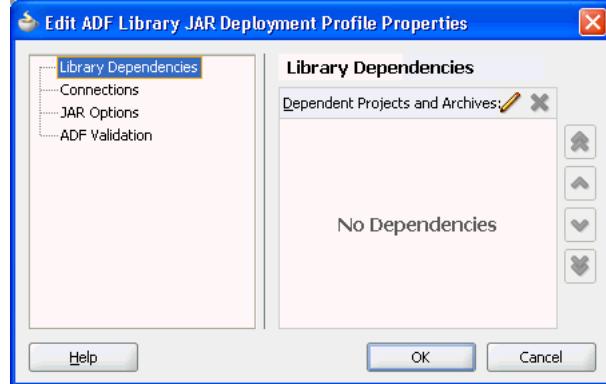
1. In the Application Navigator, double-click the project that contains the component you want to make reusable.
2. In the Project Properties dialog's left pane, select **Deployment** and then click **New**.
3. In the Create Deployment Profile dialog, select **ADF Library JAR file** for archive type, enter a name for the deployment profile, and click **OK**.

Figure 31–7 Create Deployment Profile Dialog



4. In the Project Properties dialog, select the deployment profile and click **Edit**.
5. In the Edit ADF Library JAR Deployment Profile Properties dialog, select the **Library Dependencies** node, as shown in [Figure 31–8](#).

Figure 31–8 ADF Library JAR Deployment Profile Properties Dialog

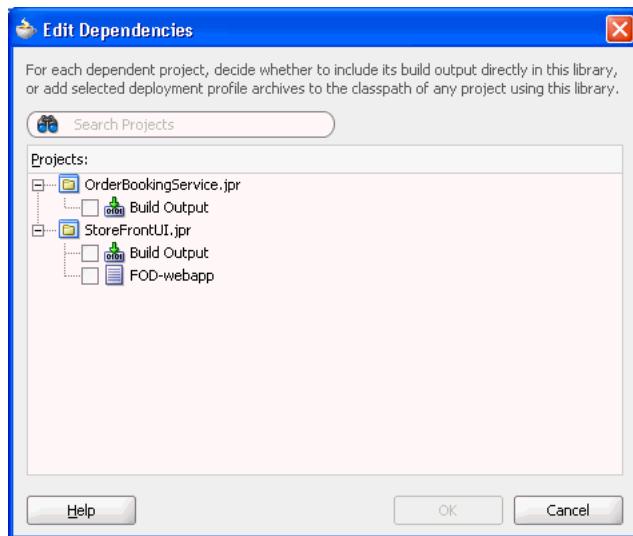


The Library Dependencies pane shows a list of dependent projects for the project being packaged. You can add the dependent project's build output directly into the packaging project's library, or you can add selected deployment profile archives to the class path.

1. To add dependent projects, click the **Edit** icon to bring up the Edit Dependencies dialog, as shown in [Figure 31–9](#).

If you select **Build Output**, the dependent project's extension libraries will be added directly to the ADF Library JAR. If you select deployment profile, the dependent project's JAR file (which includes its own extension libraries) will be added to the ADF Library JAR. For more information about library dependencies, see [Section 31.1.3, "Extension Libraries"](#).

In this example, the `OrderBookingService` project can be set as a dependency only as **Build Output**. However, the `StoreFrontUI` project can be set as a dependency either as **Build Output**, or as a deployment profile (`FOD_webapp`).

Figure 31–9 ADF Library Deployment Edit Dependencies Dialog

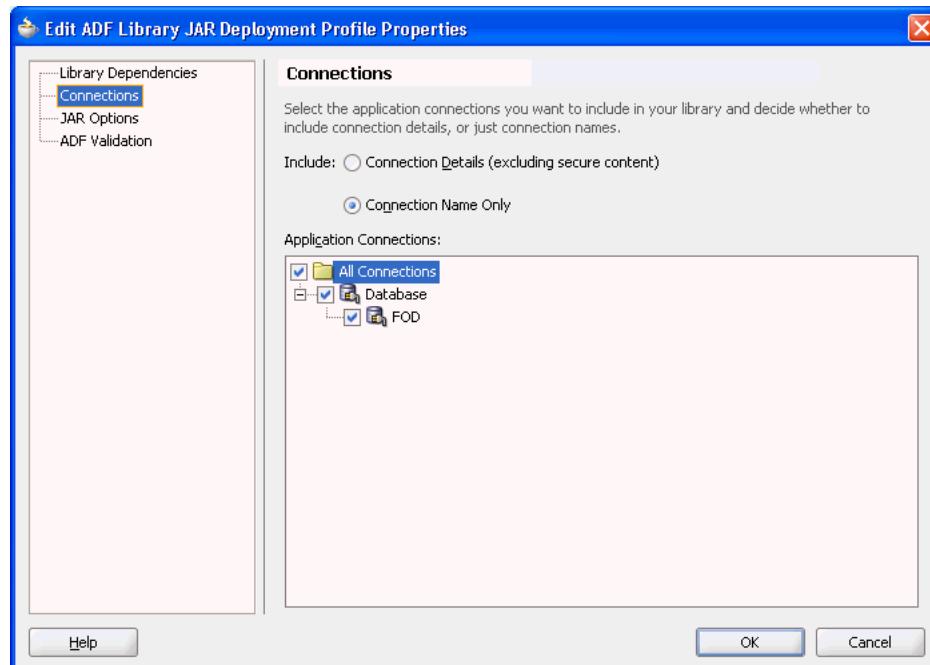
2. For each dependent project, select the **Build Output** node for the project or select the dependent profile and click **OK**.
6. In the Edit ADF Library JAR Deployment Profile Properties dialog, select the **Connections** node, as shown in [Figure 31–10](#).

You can select:

- **Connection Details (excluding secure content)**: If the project has a connection, select this checkbox if you want to add any available connection details in addition to the connection name. This is the default option. For more information, see [Section 31.1.1.6, "Including Connections Within Reusable Components"](#).
- **Connection Name Only**: Select this checkbox if you want to add the connection name without any connection details such as security.

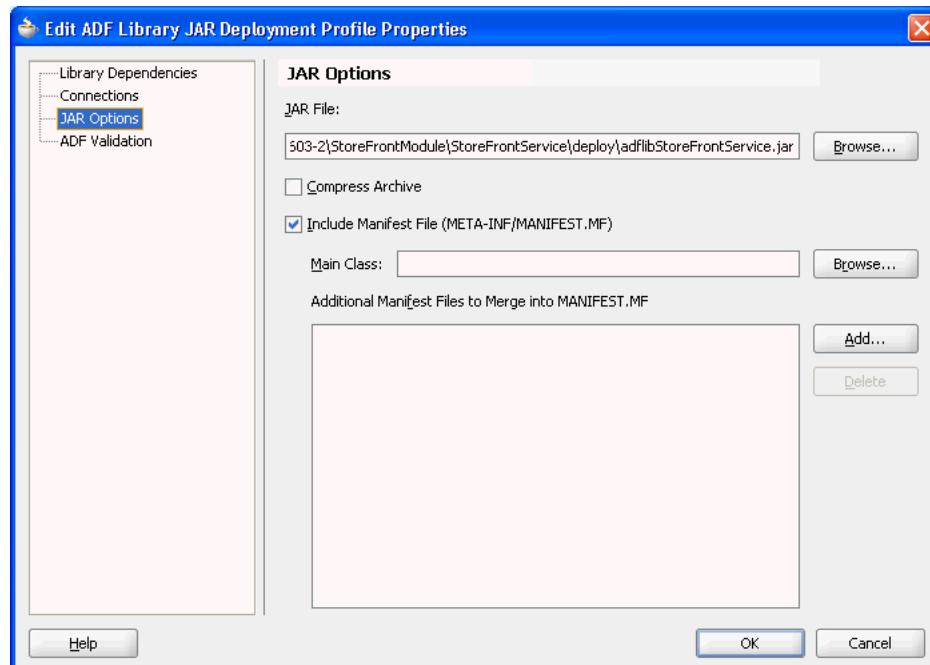
In the Applications Connections tree structure, select the checkbox for the level of connection you want to include.

Figure 31–10 Connections Page of the Edit ADF Library JAR Deployment Profile Properties Dialog



7. In the Edit ADF Library JAR Deployment Profile Properties dialog **JAR Options** node, verify the default directory path or enter a new path to store your ADF Library JAR file.

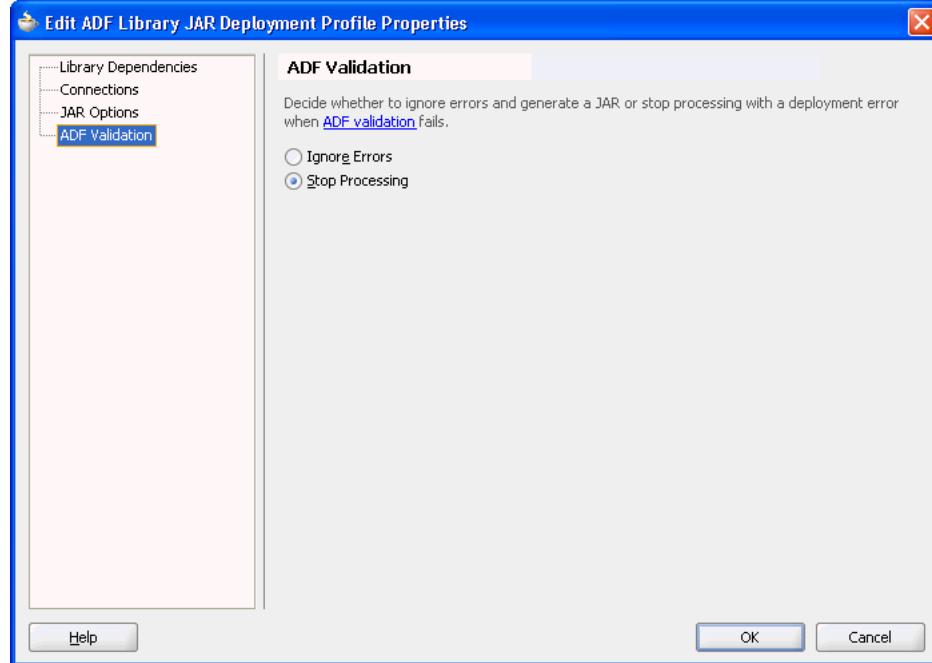
Figure 31–11 ADF Library JAR Deployment Profile Properties Dialog JAR Option



8. In the Edit ADF Library JAR Deployment Profile Properties dialog **ADF Validation** node, select:

- **Ignore Errors:** Select this option to create the JAR file even when validation fails. This is the default option.
- **Stop Processing:** Select this option to stop processing when validation fails.

Figure 31-12 ADF Library JAR Deployment Profile Properties Dialog ADF Validation



9. Click **OK** to finish setting up the deployment profile.
 10. In the Application Navigator, right-click the project and choose **Deploy > deployment > to ADF Library**, where *deployment* is the name of the deployment profile.
- JDeveloper will create the ADF Library JAR in the directory specified in Step 7. You can check that directory to see whether the JAR was created.

31.2.2 What Happens When You Package a Project to an ADF Library JAR

When you deploy the library JAR, JDeveloper packages up all the necessary artifacts, adds the appropriate control files, generates the JAR file, and places it in the directory specified in the deployment profile. During deployment, you will see compilation messages in the Log window.

When you deploy a project into an ADF Library JAR, JDeveloper performs the following actions:

- Package the HTML root directory artifacts into the JAR. When the JAR is added to the consuming project, JDeveloper will make the reusable component's `public_html` resources available by adding it to the class path.
- Add the `adfm.xml` file to the JAR. If there are multiple `META-INF/adfm.xml` files in the workspace, only the `adfm.xml` in the project being deployed is added. JDeveloper modifies this file to include relevant content from any dependent project's `adfm.xml` file.
- Add a service resources file, `oracle.adf.common.services.ResourceService.sva`, into the `META-INF`

directory of the JAR. The addition of this file differentiates an ADF Library JAR file from standard JAR files. This file defines the service strategies of the JAR and allows the Resource Palette to properly discover and display the contents of the JAR.

- Add a `Manifest.mf` file to the JAR. The `Manifest.mf` file is used to specify dependencies between JAR files, and whether to copy and include the contents of a JAR file or to reference it. JDeveloper will create a default manifest file. For example:

```
Manifest-Version: 1.0
```

- Adds a `jar-connections.xml` file to the JAR for components that require a connection and that use the connections architecture. Note that in the consuming application, connection information in configuration files that are defined in the class path and accessible at runtime may be merged together. If the same connection is named multiple times in the class path, the connection in the main application will be given priority.

Different types of reusable components have different artifact files and different entries in the service resource file.

31.2.2.1 Application Modules

For application modules, JDeveloper adds three control files to the JAR: `oracle.adf.common.services.ResourceService.sva`, `Manifest.mf`, and `adfmx.xml`. The service resource file for an application module includes entries for the Business Components associated with the application module, as well as an entry for the application module data control.

The `jar-connections.xml` file may appear for components that use the connection architecture and that contain connection information regarding the data source.

31.2.2.2 Data Controls

For data controls such as placeholder data controls, JDeveloper includes three control files in the JAR: `oracle.adf.common.services.ResourceService.sva`, `Manifest.mf`, and `DataControl.dcx` file. Data controls are used when the data source is not based on Business Components, and so Business Components are not included in the JAR file, as is the case in an application module JAR file. The service resource file for a standard data control has an entry for the data control.

The JAR also includes the `Datacontrol.dcx` file from the project to describe the data control type.

31.2.2.3 Task Flows

For task flows, JDeveloper includes three control files in the JAR: `oracle.adf.common.services.ResourceService.sva`, `Manifest.mf`, and `task-flow-registry.xml`. The service resource file for a task flow includes an entry that indicates that one or more task flows are in the JAR.

31.2.2.4 Page Templates

For page templates, JDeveloper includes two control files in the JAR: `oracle.adf.common.services.ResourceService.sva` and `Manifest.mf`.

31.2.2.5 Declarative Components

For declarative components, JDeveloper includes two control files in the JAR: `oracle.adf.common.services.ResourceService.sva` and `Manifest.mf`.

31.3 Adding ADF Library Components into Projects

After ADF Library JARs are created, they must be distributed to the developers who will use these JARs. Distributing the ADF Library JARs may include putting the JARs into network file system to be searched, browsed, and discovered. It may include using other forms of data store or services to access and retrieve these JARs. Since ADF Library JARs are simply binary files, they can be distributed like any other file such as ftp and email.

Once you have access to the ADF Library JARs, you can use JDeveloper to access them and add them to your consuming projects. Using the JDeveloper Resource Palette is the easiest and most efficient way. You can also use JDeveloper to manually add the JARs into the project by entering them into the class path.

Once reusable components have been added, how they are used depends on the type of component.

31.3.1 How to Add an ADF Library JAR into a Project using the Resource Palette

You can use the Resource Palette to search, discover, and add the ADF Library JAR to your project.

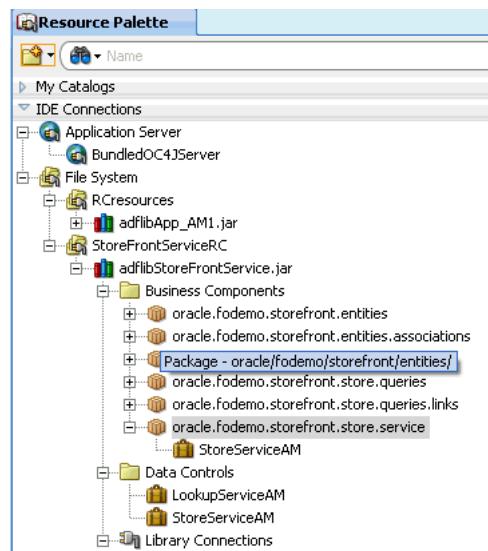
To add a component to the project using the Resource Palette:

1. From the **View** menu, choose **Resource Palette**.
2. In the Resource Palette, click the **New** icon, and then choose **New Connection > File System**.
3. In the Create File System Connection dialog, enter a name and the path of the folder that contains the JAR. For file path guidelines, see [Section 31.1.1.5, "Selecting Paths and Folders"](#).
4. Click **OK**.

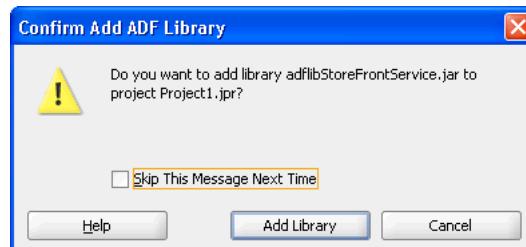
The new ADF Library JAR appears under the connection name in the Resource Palette.

5. To examine each item in the JAR structure tree, use tooltips. The tooltip shows pertinent information such as the source of the selected item.

[Figure 31–13](#) shows a Resource Palette with a tooltip message that shows package information for a business component.

Figure 31–13 Tooltip Message for a Connection in the Resource Palette

6. To add the ADF Library JAR or one of its items to the project, right-click the item and choose **Add to Project**. In the confirmation dialog that appears, click **Add Library**, as shown in [Figure 31–14](#).

Figure 31–14 Confirm Add ADF Library Dialog

If you had previously added that library JAR, you will get a Confirm Refresh ADF Library dialog asking you whether you want to refresh the library.

For application modules and data controls, you have the option to drag and drop the application module or data control from the Resource Palette into the Data Controls panel.

Note: JDeveloper will load whichever ADF Library JAR extension libraries are not already in the consuming project. Extension libraries from the ADF Library's dependent JARs will also be checked and loaded if not already part of the consuming project. For more information, see [Section 31.1.3, "Extension Libraries"](#)

31.3.2 How to Add an ADF Library JAR into a Project Manually

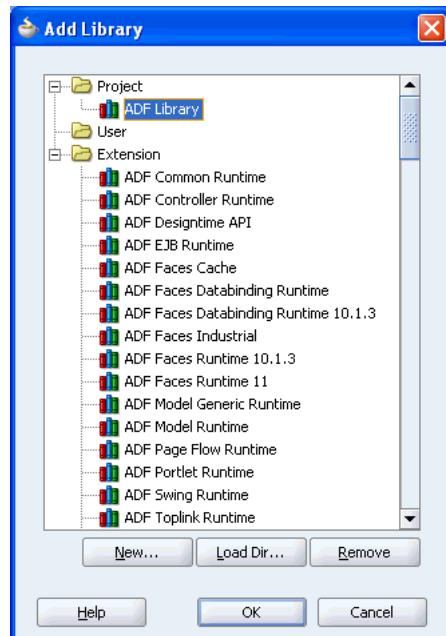
You can add an ADF Library JAR in the same way as you would other library JARs.

To add a component to a project manually:

1. In the Application Navigator, double-click the project to which the component is to be added.

2. In the Project Properties dialog, select the **Libraries and Classpath** node and then click **Add Library**.
3. In the Add Library dialog, click **New**.
4. In the Create Library dialog, select **Project** in the **Location** dropdown list and enter a name for the ADF Library. The preferable name is "ADF Library". Select **Deploy by Default**, and click **Add Entry**.
5. In the Select Path Entry dialog, enter or browse to the ADF Library JAR file and click **Select**.
6. The Create Library dialog reappears with the path of the JAR file filled in under the **Class Path** node. Click **OK**.
7. The Add Library dialog reappears with the ADF Library entry filled in under the **Project** node. Click **OK**.

Figure 31–15 Add Library Dialog



8. The Project Properties dialog reappears with the JAR file added to the list of libraries. Click **OK**.

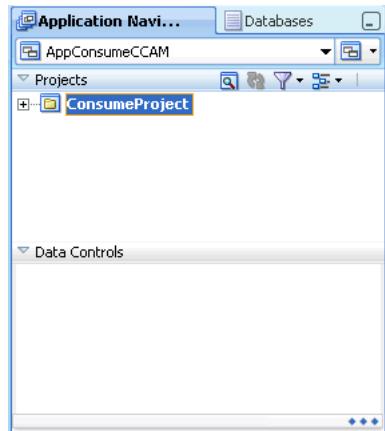
31.3.3 What Happens When You Add an ADF Library JAR to a Project

When you add an ADF Library JAR to a project, either by using the Resource Palette or by manually adding the JAR, an ADF Library definition is created in the project. The ADF Library JAR file will be added to the class path of the project. The project will access and consume the components in the JAR by reference.

By default, the **ADF Library Deployed by Default** option in the Create Library dialog is set to true. If this option is set, when the application or module is further archived or built into a WAR file, the contents of the ADF Library JAR will be copied to that archive or WAR file. If the Deploy by Default option is not set, then the JARs in the ADF Library must be loaded in some other way, such as by deploying them in a shared library.

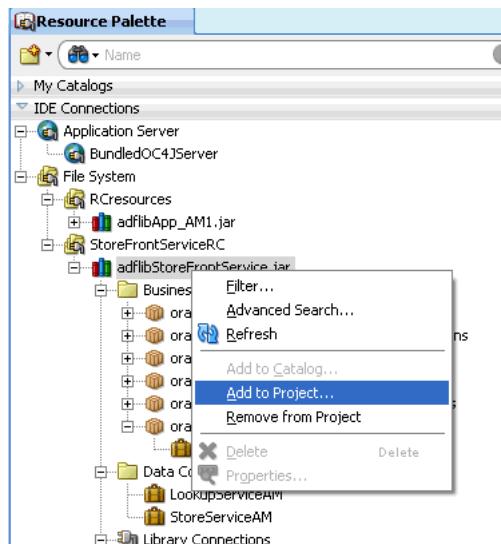
[Figure 31–16](#) shows the empty Data Controls panel for a consuming project before the ADF Library was added.

Figure 31–16 Data Controls Panel of the Consuming Project

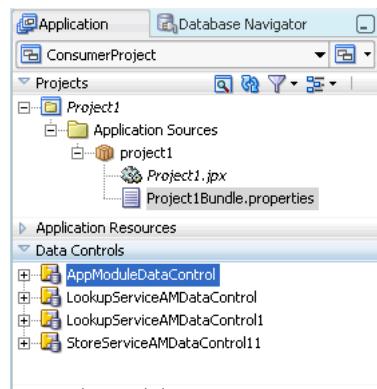


[Figure 31–17](#) shows the adflibStoreFrontService ADF Library being added to the consuming project.

Figure 31–17 Adding the ADF Library JAR to the Project



[Figure 31–18](#) shows several data controls from the ADF Library added to the Data Controls panel in the consuming project.

Figure 31–18 Consuming Project Data Controls Panel with Added Application Modules

After adding the ADF Library JAR, you may notice some changes to some of the JDeveloper windows. These changes are different depending on the type of components being added. [Table 31–3](#) lists the effects on several JDeveloper windows.

Table 31–3 JDeveloper Window After Adding an ADF Library

Added Component	Data Controls Panel	Creation Wizards	Component Palette
Data controls	Data control appears.		
Application module	Application module appears.		
Business Components		Entity objects available in view object creation wizard. View objects in JAR also available for use.	
Task flows			Task flows appear in Component Palette.
Page template		Page template available during JSF creation wizard.	
Declarative components			Tag library appears in Component Palette. Declarative component appears in the list.

31.3.4 What You May Need to Know About Using ADF Library Components

Although the procedure to add an ADF Library JAR to a project is standardized, the component type determines where it appears in JDeveloper and how it can be reused.

31.3.4.1 Using Data Controls

When you add a data control to a project, the data control appears in the Data Controls panel. If you are using the Resource Palette, you have the option of dragging and dropping the data control from the Resource Palette onto the Data Controls panel, and then dragging and dropping from the Data Controls panel onto the page.

31.3.4.2 Using Application Modules

When you add an application module to a project, the application module appears in the Data Controls panel. If you are using the Resource Palette, you have the option of dragging and dropping the application module item from the Resource Palette onto the Data Controls panel, and then dragging and dropping from the Data Controls panel onto the page.

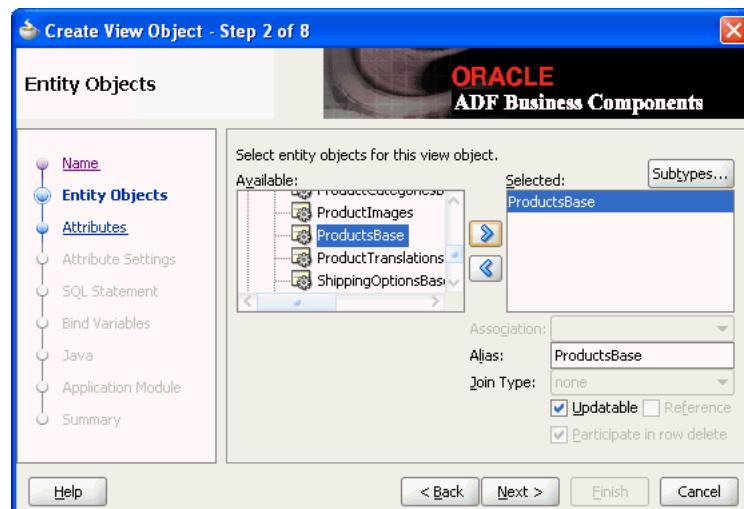
Application modules are associated with Business Components. When the reusable application module was packaged, the JAR includes the Business Components used to create the application module. These components will be available for reuse.

31.3.4.3 Using Business Components

Business components can also be packaged and reused. The entity objects, view objects, and associations can be packaged together and added to a consuming project. By default, packaged application modules will include the Business Components in the JAR, but Business Components can be reused by themselves without the accompanying application module.

One way to reuse Business Components is to create new view objects using the entity objects from an ADF Library JAR. When you add view objects using the wizard, the entity objects will become available within the wizard to support view object generation. For instructions on creating view objects, see [Section 5.2.1, "How to Create an Entity-Based View Object"](#). When the wizard presents a screen for entity objects used to create view objects, the entity objects from the ADF Library will be available in the shuttle window, as shown in [Figure 31–19](#).

Figure 31–19 Creating View Object Using Entity Objects from ADF Library



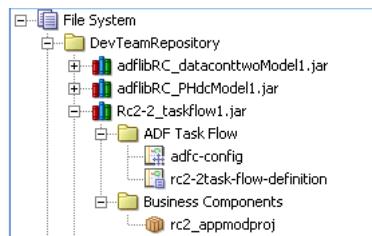
If the consuming project has the **Initialize Project for Business Component** option selected (in the Project Properties dialog Business Components page) before the ADF Library JAR is added, the Business Components within the JAR will automatically be made available to the project.

If you add the ADF Library JAR first, and then select **Initialize Project for Business Component**, JDeveloper will automatically load the Business Components.

31.3.4.4 Using Task Flows

Task flows added to a project will appear in the Component Palette when you are adding components to a JSF page. If you are using the Resource Palette, you can also drag and drop the task flow directly from the Resource Palette onto another task flow or page, as shown in [Figure 31–20](#).

Figure 31–20 Using the Resource Palette to Drag and Drop Task Flows

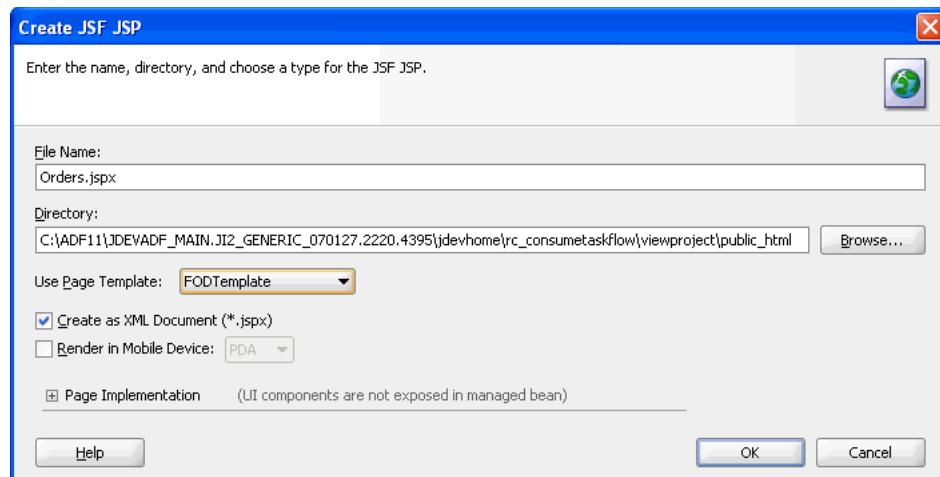


For more information about creating task flows, see [Chapter 13, "Getting Started with ADF Task Flows"](#).

31.3.4.5 Using Page Templates

When you add a page template to a project, the template will not be exposed in the Application Navigator. You will not have direct access to individual supporting files, such as image files. However, the template retains its access to its supporting files inside the JAR and is fully reusable within the project. When you apply the template, it will retain all the images that were loaded with the template.

The page template is exposed and accessible when you create a new JSF JSP page using the wizard. When the wizard presents you with the option to use a page template, the ADF Library template will appear in the dropdown list. For example, if you loaded a page template called FODTemplate from an ADF Library, when you use the wizard to create a JSF JSP page, FODTemplate will appear in the wizard, as shown in [Figure 31–21](#). For information on how to use page templates and create a JSF JSP page, see [Section 18.2, "Using Page Templates"](#) and the *Oracle Fusion Middleware Web User Interface Developer's Guide for Oracle Application Development Framework*.

Figure 31–21 Using a Page Template from ADF Library

31.3.4.6 Using Declarative Components

When you add a declarative component to a project, the JSP tag libraries that contain the component will be added to the project. The tag libraries will appear in the Component Palette, and the declarative components will be available for selection. For information about creating and using declarative components, see the *Oracle Fusion Middleware Web User Interface Developer's Guide for Oracle Application Development Framework*.

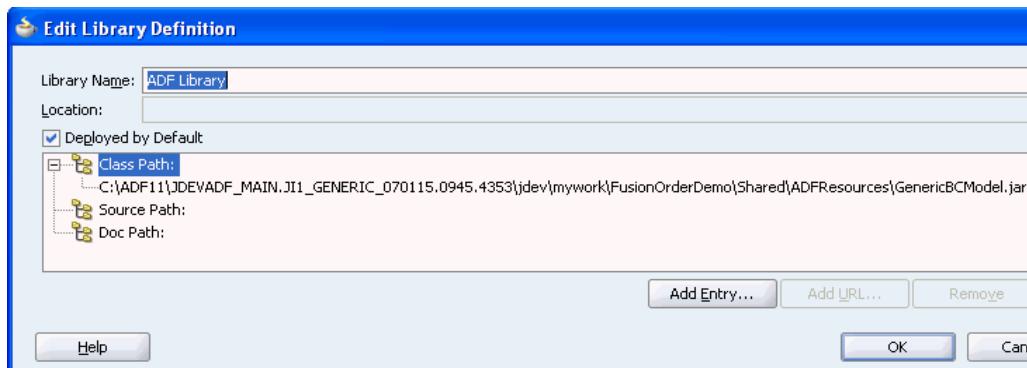
31.3.5 What You May Need to Know About Differentiating ADF Library Components

If you mix components created during application development with components imported from the ADF Library, you may be able to differentiate between them by using the tooltips feature of JDeveloper.

Move the cursor over an application module or data control and you will see the full path of the source. If you see the ADF Library JAR file in the path, that means the component source is the ADF Library.

31.3.6 What Happens at Runtime: Adding ADF Libraries

After an ADF Library JAR has been added to a project and to the class path, it behaves like any other library file. During runtime, any component that uses the component in the ADF Library JAR will reference that object. The process is transparent and there is no need to distinguish between components that were developed for the project and those that are in ADF Library JARs. Figure 31–22 shows the ADF Library and the path to the JAR as defined for a project.

Figure 31–22 Edit Library Definition dialog showing the ADF Library in the Class Path

31.4 Removing an ADF Library JAR from a Project

You can use the Resource Palette to remove an ADF Library JAR from a project, or you can manually remove the JAR using the Project Properties dialog. You can remove an ADF Library JAR only if the components in the project do not have any dependencies on the components in the ADF Library JAR.

When you remove a JAR, it will no longer be in the project class path and its components will no longer be available for use.

31.4.1 How to Remove an ADF Library JAR from a Project Using the Resource Palette

The Resource Palette allows you to remove previously added ADF Library JARs from a project using a simple command.

To remove an ADF Library JAR from the project using the Resource Palette:

1. From the **View** menu, choose **Resource Palette**.
2. In the Application Navigator, select the project that has the ADF Library JAR you want to remove.
3. In the Resource Palette, locate the ADF Library JAR you want to remove from the current project.
4. Right-click the JAR and choose **Remove from Project**.

31.4.2 How to Remove an ADF Library JAR from a Project Manually

When you remove an ADF Library JAR manually, be sure to remove the correct library. Be aware of other libraries that are critical to the operation of the project.

To remove an ADF Library JAR from the project manually:

1. In the Application Navigator, double-click the project.
2. In the Project Properties dialog, select the **Libraries and Classpath** node.
3. In the **Classpath Entries** list, select **ADF Library** and click **Edit**.
4. In the Edit Library Definition dialog, select the ADF Library JAR you want to remove under the **Class Path** node, and click **Remove**.
5. Click **OK** to accept the deletion, and click **OK** again to exit the dialog.

32

Deploying Fusion Web Applications

This chapter describes how to deploy Oracle ADF applications to a target application server. It describes how to create deployment profiles, deployment descriptors, and how to load ADF runtime libraries. It also contains information on deploying to Oracle WebLogic Server 10gR3.

This chapter includes the following sections:

- [Section 32.1, "Introduction to Deploying Fusion Web Applications"](#)
- [Section 32.2, "Creating a Connection to the Target Application Server"](#)
- [Section 32.3, "Creating a Deployment Profile"](#)
- [Section 32.4, "Creating and Editing Deployment Descriptors"](#)
- [Section 32.5, "Installing the ADF Runtime to the WebLogic Installation"](#)
- [Section 32.6, "Creating and Extending WebLogic Domains"](#)
- [Section 32.7, "Creating a JDBC Data Source for a WebLogic Domain Server"](#)
- [Section 32.8, "Deploying the Application"](#)
- [Section 32.9, "Testing the Application and Verifying Deployment"](#)

32.1 Introduction to Deploying Fusion Web Applications

Deployment is the process of packaging application files as an archive file and transferring it to a target application server. You can use JDeveloper to deploy Oracle ADF applications directly to Oracle WebLogic Server 10gR3, or indirectly to an archive file as the deployment target, and then install this archive file to the target server. For application development, you can also use JDeveloper to run an application in the Integrated WLS Server.

Oracle WebLogic Server 10gR3 Java EE applications are based on standardized, modular components. Oracle WebLogic Server 10gR3 provides a complete set of services for those modules and handles many details of application behavior automatically, without requiring programming.

Deploying a Fusion web application is slightly different from deploying a standard Java EE application.

JSF applications that contain ADF Faces components have a few additional deployment requirements:

- ADF Faces require Sun's JSF Reference Implementation 1.2 and MyFaces 1.0.8 (or later).

- ADF Faces applications cannot run on an application server that supports only JSF 1.0.

You can use JDeveloper to:

- Deploy to an Oracle WebLogic Server 10gR3

You can deploy applications directly to a Oracle WebLogic Server.

- Deploy to an archive file

You can deploy applications indirectly by choosing an EAR file as the deployment target. The archive file can subsequently be installed on a target Oracle WebLogic Server 10gR3.

- Run applications in Integrated WLS

You can run and debug applications using the Integrated WLS. The WLS instances can be bound to an application and then started and stopped.

If you are developing an application in JDeveloper and want to run the application in the Integrated WLS, you do not need to perform the tasks required for deploying directly to Oracle WebLogic Server or to an archive file. JDeveloper has a default connection to the Integrated WLS and does not require any deployment profiles or descriptors. The Integrated WLS also has access to the ADF libraries required to run an ADF application. You can run an application by selecting **Run** from the JDeveloper menu. You also debug the application using the features described in [Chapter 29, "Testing and Debugging ADF Components"](#).

In general (other than Integrated WLS), you use JDeveloper to prepare the application or project for deployment by:

- Creating a connection to the target application server
- Creating a deployment profile
- Creating deployment descriptors
- Updating `application.xml` and `web.xml` to be compatible with Oracle WebLogic Server

You also must prepare Oracle WebLogic Server for ADF application deployment by:

- Installing the ADF Runtime into the Oracle WebLogic Server installation (if it is not already installed)
- Extending a domain to be ADF compatible using the ADF Runtime
- Creating a global JDBC data source for applications that require a connection to a data source
- Migrating application-level policy data to a domain-level policy store.

After the application and Oracle WebLogic Server have been prepared, you can:

- Use JDeveloper to:
 - Directly deploy to Oracle WebLogic Server using the deployment profile and the application server connection
 - Deploy to an EAR file using the deployment profile
- Using the Oracle WebLogic Server Administration Console to deploy the EAR file created in JDeveloper

32.2 Creating a Connection to the Target Application Server

You can deploy applications to Oracle WebLogic Server via JDeveloper application server connections.

To create a connection to an application server:

1. Launch the Application Server Connection wizard.

You can:

- In the Application Server Navigator, right-click **Application Servers** and choose **New Application Server Connection**.
 - In the New Gallery, expand **General**, select **Connections** and then **Application Server Connection**, and click **OK**.
 - In the Resource Palette, choose **New > New Connections > Application Server**.
2. In the Name and Type page of the Create AppServer Connection dialog, enter a connection name.
 3. In the **Connection Type** dropdown list, choose **WebLogic 10.3** to create a connection to Oracle WebLogic Server 10gR3.
 4. Click **Next**.

5. On the Authentication page, enter a user name and password for the administrative user authorized to access the Oracle WebLogic Server.
6. Select **Deploy Password** to include the password for this connection with the archives deployed for it. Deselect it to require the user to supply a password when connecting.

Typically, you would select this option for development testing purposes only when using the Integrated WLS server in JDeveloper, because it enables you to connect more quickly. You should then deselect it again before you deploy your final application.

7. Click **Next**.
8. On the Configuration page, enter a Oracle WebLogic host name.
The Oracle WebLogic host name is the name of the Oracle WebLogic Server containing the TCP/IP DNS where your application (`.jar`, `.war`, `.ear`) will be deployed.
9. In the **Port** field, enter a port number for the Oracle WebLogic Server on which your application (`.jar`, `.war`, `.ear`) will be deployed.
If you don't specify a port, the port number defaults to 7001.
10. In the **SSL Port** field, enter an SSL port number for the Oracle WebLogic Server on which your application (`.jar`, `.war`, `.ear`) will be deployed.

Specifying an SSL port is optional. It is required only if you want to ensure a secure connection for deployment.

- If you don't specify an SSL port, the port number defaults to 7002.
11. Select **Always use SSL** to connect to the Oracle WebLogic Server using the SSL port.
 12. Optionally enter a **WLS Domain** only if the Oracle WebLogic Server is configured to distinguish nonadministrative server nodes by name

13. Click **Next**.
14. On the Test page, click **Test Connection** to test the connection.
JDeveloper performs several types of connections tests. The JSR-88 test must pass for the application to be deployable. If the test fails, return to the previous pages of the wizard to fix the configuration.
15. Click **Finish**.

32.3 Creating a Deployment Profile

A *deployment profile* defines the way the application is packaged into the archive that will be deployed to the target environment. The deployment profile:

- Specifies the format and contents of the archive file that will be created
- Lists the source files, deployment descriptors, and other auxiliary files that will be packaged
- Describes the type and name of the archive file to be created
- Highlights dependency information, platform-specific instructions, and other information

You should create a WAR deployment profile for each web (ViewController) project that you want to deploy in your application. Then you create an application-level EAR deployment profile and select the projects you want to include from a list. When the application is deployed, the EAR file will include all the projects that were selected in the deployment profile.

32.3.1 How to Create Deployment Profiles

Before you create project and application deployment profiles, you must have already created a Fusion web application.

To create deployment profiles for an application:

1. In the Application Navigator, right-click the web project that you want to deploy and choose **New**.
2. In the New Gallery, expand **General** and select **Deployment Profiles**.
If you don't see **Deployment Profiles** in the Categories tree, click the **All Technologies** tab.
3. In the **Items** list, choose **WAR File** and click **OK**.
4. In the Create Deployment Profile -- WAR File dialog, enter a name for the project deployment profile and click **OK**.
5. In the Edit WAR Deployment Profile Properties dialog, choose items in the left pane to open dialog pages in the right pane. Configure the profile by setting property values in the pages of the dialog.

Note: You may want to change the Java EE web context root setting (choose **General** in the left pane).

By default, when **Use Project's Java EE Web Context Root** is selected, the associated value is set to the project name, for example, Application1-Project1-context-root. You need to change this if you want users to use a different name to access the application.

If you are using custom JAAS LoginModule for authentication with JAZN, the context root name also defines the application name that is used to look up the JAAS LoginModule.

6. Click **OK** to exit the Deployment Profile Properties dialog.
7. Click **OK** again to exit the Project Properties dialog.
8. Repeat Steps 1 through 7 for all web projects that you want to deploy.
9. In the Application Navigator, right-click the application and click **New**.
10. In the New Gallery, expand **General** and choose **Deployment Profiles**.
If you don't see **Deployment Profiles** in the Categories tree, click the **All Technologies** tab.
11. In the **Items** list, choose **EAR File** and click **OK**.
12. In the Create Deployment Profile -- EAR File dialog, enter a name for the application deployment profile and click **OK**.
13. In the Edit EAR Deployment Profile Properties dialog, choose items in the left pane to open dialog pages in the right pane. Configure the profile by setting property values in the pages of the dialog.

Be sure that you:

- Select **Application Assembly** and then in the **Java EE Modules** list, select all the project profiles that you want to include in the deployment.
- Select **Platform**, select **WebLogic 10.3** as the default platform and then select the target application connection from the **Target Connection** dropdown list.

Note: If you are using custom JAAS LoginModule for authentication with JAZN, the context root name also defines the application name that is used to look up the JAAS LoginModule.

14. Click **OK** to exit the Deployment Profile Properties dialog.
15. Click **OK** again to exit the Application Properties dialog.

32.3.2 How to View and Change Deployment Profile Properties

After you have created a deployment profile, you can view and change its properties.

To view, edit, or delete a project's deployment profile:

1. In the Application Navigator, right-click the project and choose **Project Properties**.
2. In the Project Properties dialog, click **Deployment**.

The **Deployment Profiles** list displays all profiles currently defined for the project.

3. In the list, select a deployment profile.
4. To edit or delete a deployment profile, click **Edit** or **Delete**.

32.4 Creating and Editing Deployment Descriptors

Deployment descriptors are server configuration files that define the configuration of an application for deployment and are deployed with the Java EE application as needed. The deployment descriptors that a project requires depend on the technologies the project uses and on the type of the target application server. Deployment descriptors are XML files that can be created and edited as source files, but for most descriptor types, JDeveloper provides dialogs that you can use to view and set properties. If you cannot edit these files declaratively, JDeveloper opens the XML file in the source editor for you to edit its contents.

You can specify deployment descriptors that are specific to your target Oracle WebLogic application service.

In addition to the standard Java EE deployment descriptors (for example, `application.xml` and `web.xml`), you can also have deployment descriptors that are specific to your target application server. For example, if you are deploying to Oracle WebLogic Server, you can also have `weblogic.xml`, `weblogic-application.xml`, and `weblogic-ejb-jar.xml`.

In the application EAR file, make sure it includes a `weblogic-application.xml` file and that the file contains a reference to `adf.oracle.domain`. It should have a `<library-ref>` entry as shown in [Example 32–1](#).

Example 32–1 library-ref tag in weblogic-application.xml

```
<library-ref>
    <library-name>adf.oracle.domain</library-name>
</library-ref>
```

Because Oracle WebLogic Server 10gR3 runs on Java EE 1.5, you may need to modify the `application.xml` and `web.xml` to be compatible with Oracle WebLogic Server.

32.4.1 How to Create Deployment Descriptors

Before you create a deployment descriptor, you should check to see if JDeveloper has already generated it.

To create a deployment descriptor:

1. In the Application Navigator, right-click the project for which you want to create a descriptor and choose **New**.
2. In the New Gallery, expand **General** and choose **Deployment Descriptors**.
3. In the **Items** list, choose a descriptor type, and click **OK**.

If you can't find the item you want, make sure that you chose the correct project, and chose **All Technologies** tab or use the Search field to find the descriptor. If the item is not enabled, check to make sure the project does not already have a descriptor of that type. A project may have only one instance of a descriptor.

JDeveloper starts the Create Deployment Descriptor wizard or opens the file in the source editor, depending on the type of deployment descriptor you choose.

Note: For EAR files, do not create more than one deployment descriptor per application or workspace. These files are assigned to projects, but have application workspace scope. If multiple projects in an application have the same deployment descriptor, the one belonging to the launched project will supersede the others. This restriction applies to `application.xml`, `weblogic-jdbc.xml`, `jazn-data.xml`, and `weblogic.xml`.

The best place to create an application-level descriptor is in the **Descriptors** node of the Application Resources panel in the Application Navigator. This ensures that the application is created with the correct descriptors.

32.4.2 How to View or Change Deployment Descriptor Properties

After you have created a deployment descriptor, you can change its properties using JDeveloper dialogs or by editing the file in the source editor. The deployment descriptor is an XML file (for example, `application.xml`) typically located under the **Application Sources** node.

To view or change deployment descriptor properties:

1. In the Application Navigator, right-click the deployment descriptor and choose **Properties**.

The Properties dialog that appears depends on the selected descriptor type.

If the context menu does not have a **Properties** item, then the descriptor must be edited as a source file. Choose **Open** from the context menu to open the profile in the source editor.

2. In the Properties dialog, choose items in the left pane to open dialog pages in the right pane. Configure the descriptor by setting property values in the pages of the dialog.
3. Click **OK**.

32.4.3 How to Create or Configure the `application.xml` file for WebLogic Compatibility

You may need to configure your `application.xml` file to be compliant with Java EE 1.5.

Note: Typically, your project has an `application.xml` file that is compatible and you would not need to perform this procedure.

To create the `application.xml` file:

1. In the Application Navigator, right-click the project and choose **New**.
2. In the New Gallery, expand **General**, choose **Deployment Descriptors**, and in **Items**, choose **Java EE Deployment Descriptor Wizard** and click **OK**.
3. In the Select Descriptor page of the Create Java EE Deployment Descriptor dialog, choose `application.xml` and click **Next**.
4. In the Select Version page, choose **5.0** and click **Next**.
5. In the Summary page, click **Finish**.

32.4.4 How to Create or Configure the web.xml file for WebLogic Compatibility

You may need to configure your web.xml file to be compliant with Java EE 1.5 (which corresponds to servlet 2.5 and JSP 1.2).

Note: Typically, your project has a web.xml file that is compatible and you would not need to perform this procedure.

To create the web.xml file:

1. In the Application Navigator, right-click the project and choose **New**.
2. In the New Gallery, expand **General**, choose **Deployment Descriptors**, and in **Items**, choose **Java EE Deployment Descriptor Wizard** and click **OK**.
3. In the Select Descriptor page of the Create Java EE Deployment Descriptor dialog, choose **web.xml** and click **Next**.
4. In the Select Version page, choose **2.5** and click **Next**.
5. In the Summary page, click **Finish**.

32.4.5 How to Create Deployment Descriptors for Applications with ADF Faces Components

If your application uses ADF Faces components, ensure that the standard Java EE deployment descriptors contain entries for ADF Faces and that you include the ADF and JSF configuration files in your archive file (typically a WAR file). When you create ADF Faces components in your application, JDeveloper automatically creates and configures the files for you.

If you are using ADF databound UI components as described in [Section 11.3, "Using the Data Controls Panel"](#), ensure that you have the `DataBindings.cpx` file. For information about the file, see [Section 11.4, "Working with the DataBindings.cpx File"](#).

32.5 Installing the ADF Runtime to the WebLogic Installation

The Oracle WebLogic Server requires the ADF Runtime to run Oracle ADF applications.

Installing the ADF Runtime is not required if you are using JDeveloper to run applications in the Integrated WLS.

If you installed the Oracle WebLogic Server 10gR3 together with JDeveloper 11g using the Oracle Installer (or installed it with the **Application Development Framework Runtime** option selected), the ADF Runtime is installed as part of the Oracle WebLogic Server installation. You can proceed to [Section 32.6, "Creating and Extending WebLogic Domains"](#).

If you already have an existing Oracle WebLogic Server 10gR3 installation, you can add the ADF Runtime into that installation using the Oracle Installer

32.5.1 How to Install the ADF Runtime into an Existing WebLogic Server

Before you add the ADF Runtime into an existing Oracle WebLogic Server 10gR3 installation, you must have obtained the Oracle Installer. You can download the installer from the Oracle Technology Network (OTN) web site at <http://www.oracle.com/technology/software/products/jdev/index.html>.

Use the instructions in the *Oracle JDeveloper 11g Installation Guide* to start the installer and to complete the installation.

To install the ADF Runtime into an existing WebLogic Server:

1. Start the Oracle Installer as described in the *Oracle JDeveloper 11g Installation Guide*.
2. In the Choose Middleware Home directory page, select **Use an existing Middleware Home**, select the directory in which your Oracle WebLogic Server 10gR3 resides, and click **Next**.
3. In the Choose Products and Components page, choose **Application Development Framework Runtime**, (deselect **JDeveloper Studio** if you do not want to install the JDeveloper IDE) and click **Next**.

The Oracle WebLogic Server product and components should be gray and unselectable. If it is enabled and selected, you must check whether you have selected the correct Oracle WebLogic Server installation directory.

4. Follow the instructions in the *Oracle JDeveloper 11g Installation Guide* to complete the installation
5. When the installation is complete, click **Done**.

Follow the instructions in [Section 32.6, "Creating and Extending WebLogic Domains"](#) to use the Oracle WebLogic Configuration Wizard to create or extend a WebLogic domain.

32.5.2 What You May Need to Know About ADF Libraries

A JAR file is an ADF Library if it contains JAR services registered for ADF components such as ADF task flows, pages, or application modules. If the JAR file contains these components, the ADF Library provides summary information that can be browsed in the Resource Palette. For more information, see [Chapter 31, "Reusing Application Components"](#).

32.6 Creating and Extending WebLogic Domains

You need to create or configure a WebLogic domain to accept ADF applications. If you do not already have a domain, you need to create a new domain. If you already have a domain, you must extend the domain before it can run ADF applications.

If you are using WebLogic Managed servers to run your applications, you may need to configure your Managed server. For more information about configuring a WebLogic Managed server on Oracle WebLogic Server, see the "Customizing the Environment" chapter in the *Creating WebLogic Domains Using the Configuration Wizard* guide on the Oracle Technology Network website at

http://download.oracle.com/docs/cd/E12839_01/common/docs103/configwiz/custom.html.

32.6.1 How to Create a WebLogic Domain for Oracle ADF

You must create a WebLogic domain if it does not already exist.

To create a new WebLogic domain:

1. Start the configuration by choosing **Oracle Fusion Middleware 11.1.1.0.1 > WebLogic Server 10.3 > Tools > Configuration Wizard** from the Windows **Start** menu.

Or, change to the ORACLE_HOME/wlserver_10.3/common/bin directory and run config.cmd or config.sh (depending on your platform) from the command line.

2. In the Welcome page, select **Create a New WebLogic Domain** and click **Next**.
3. In the Select Domain Source page, select **Generate a domain configured automatically to support the following products**, then select **Application Development Framework** and click **Next**.
The option **WebLogic Server (Required)** is already selected.
4. In the Configure Administrator Username and Password page, enter the user name and password and click **Next**.
5. In the Configure Server Start Mode and JDK page, select whether the domain is for a development or a production system, choose the JDK to use, and click **Next**.
6. In the Customize Environment and Services Setting page, select **No** when asked to customize further domain options and click **Next**.
7. In the Create WebLogic Domain page, enter the name and location of the new domain, the location of applications, and click **Create**.
8. Click **Done** when the creation process has finished.

32.6.2 How to Extend a WebLogic Domain for Oracle ADF

If you have an existing WebLogic domain that has not been updated to run Oracle ADF applications, you can extend it for Oracle ADF. You must already have the ADF Runtime installed in the WebLogic Server installation.

To extend a WebLogic Domain for Oracle ADF:

1. Start the configuration by choosing **Oracle Fusion Middleware 11.1.1.0.1 > WebLogic Server 10.3 > Tools > Configuration Wizard** from the Windows **Start** menu.
Or, change to the ORACLE_HOME/wlserver_10.3/common/bin directory and run config.cmd or config.sh (depending on your platform) from the command line.
2. In the Welcome page, select **Extend an existing WebLogic domain** and click **Next**.
3. In the Select a WebLogic Domain Directory page, select the location of the domain you want to configure for ADF, and click **Next**.
4. In the Select Extension Source page, select **Extend my domain automatically to support the following added products**, then select **Application Development Framework** and click **Next**.
5. In the Customize JDBC and JMS File Store Settings page, leave **No** selected to update the domain as specified in the template and click **Next**.
6. In the Extend WebLogic Domain page, enter the application location and then click **Extend**.
7. Click **Done** when the configuration has finished.
8. After you have extended the domain, check the POST_CLASSPATH entry to make sure that setDomainEnv is set to the correct location.

The correct location should be ORACLE_HOME\jdeveloper\modules\features\adf.share_11.1.1.jar.

This configures the rest of the runtime .jar files using the manifest file.

Note: Your application's EAR file must have a `weblogic-application.xml` file containing a reference to the `adf.oracle.domain` shared library.

You can now start the Oracle WebLogic Server by running the command line script `ORACLE_HOME\user_projects\domains\domain_name\bin\startWebLogic.cmd` and stop the server using the `stopWebLogic.cmd` script in the same directory.

Access the Oracle WebLogic Server Administration Console using the URL `http://localhost:7001/console`.

32.7 Creating a JDBC Data Source for a WebLogic Domain Server

Oracle ADF applications can use either a data source or JDBC URL for database connections. You use the Oracle WebLogic Server Administration Console to configure a JDBC data source.

To configure a WebLogic domain server for both data source and JDBC URL:

1. Start the Oracle WebLogic Server (if not already started) by choosing **Oracle Fusion Middleware 11.1.1.0.1 > User Projects > Domain > Start Admin Server for WebLogic Server Domain** from the Windows **Start** menu.
2. Start the Oracle WebLogic Server Administration Console by choosing **Oracle Fusion Middleware 11.1.1.0.1 > User Projects > Domain > Admin Server Console** from the Windows **Start** menu.
3. Log in to the Oracle WebLogic Server Administration Console.
4. In the WebLogic Server Administration Console page, select **JDBC > Data Sources**.
5. Click **New**.
6. In the JDBC Data Source Properties page:
 - In the **Name** field, enter the name of the JDBC data source.
 - In the **JNDI** field, enter the name of the connection in the form `jdbc/connection DS`.
 - For the **Database Type**, choose **Oracle**.
 - For the **Database Driver**, choose **Oracle Driver (thin)**, and click **Next**.
7. In the Transactions Options page, accept the default options and click **Next**.
8. In the Connection Properties page:
 - For **Database Name**, enter the Oracle SID. For example, `orcl`.
 - For **Host Name**, enter the machine name of the database.
 - Enter the port number used to access the database.
 - Enter the user name and password for the database and click **Next**.
9. In the Test Database Connection page, click **Test Configuration** to test the connection.

10. In the Select Targets page, select the server for which the JDBC data source is to be deployed.
11. Click **Finish**.

Once the data source has been created on the domain server, it can be used by an application module.

32.8 Deploying the Application

Before you can deploy the application to your target application server, you may need to perform some vendor-specific configuration. See your application server documentation for instructions.

You can deploy directly to Oracle WebLogic Server if you have set up a connection in JDeveloper to your Oracle WebLogic Server.

Note: When you are deploying to Oracle WebLogic Server 10gR3 from JDeveloper, ensure that the HTTP Tunneling property is enabled in the Oracle WebLogic Server Administration Console. This property is located under **Servers > *ServerName* > Protocols**. *ServerName* refers to the name of your Oracle WebLogic Server.

[Table 32–1](#) shows the supported version of Oracle WebLogic Server.

Table 32–1 Support for Oracle WebLogic Server

WebLogic version	JDK version	Java EE version
10.3	1.6	Java EE 1.5

Before deploying applications that use Oracle ADF to Oracle WebLogic Server, you need to install the ADF Runtime on the server. For more information, see [Section 32.5, "Installing the ADF Runtime to the WebLogic Installation"](#).

[Table 32–2](#) describes some common deployment techniques that you can use during the application development and deployment cycle. The deployment techniques are listed in order from deploying on development environments to deploying on production environments. It is likely that in the production environment, the system administrators deploy applications using scripting tools.

Table 32–2 Deployment Techniques

Deployment Technique	When to Use
Run directly from Oracle JDeveloper	<p>When you are developing your application. You may want to deploy the application quickly for testing. You want deployment to be quick because you will be repeating the editing and deploying process many times.</p> <p>Oracle JDeveloper contains Integrated WLS, on which you can run and test your application. You should also deploy your application to an external application server to test it.</p>
Deploy to EAR file, then use the target application server's tools for deployment	<p>When you are ready to deploy and test your application on an application server in a test environment. On the test server, you can test features (such as LDAP and OracleAS Single Sign-On Server) that are not available on the development server.</p> <p>You can also use the test environment to develop your deployment scripts, for example, using Ant.</p>
Use a script to deploy applications	<p>When your application is in a test and production environment. In production environments, system administrators usually run scripts to deploy applications.</p>

Instead of deploying applications directly from JDeveloper, you can use JDeveloper to create the archive file, and then deploy the archive file using the Oracle WebLogic Server Administration Console.

If your application has security credential data, including database credentials, you must migrate your application-level policy data to a domain-level policy store. For more information, see "Migrating the Security Repository to a Production Environment" in the "Developing Secure Applications" section of the JDeveloper online help

32.8.1 How to Deploy to a WebLogic Server from JDeveloper

Before you deploy an application or project, you should have already created an application-level deployment profile that deploys to an EAR file.

To deploy to the target application server from JDeveloper:

- To deploy an application, in the Application Navigator, right-click the application and choose **Deploy > deployment profile > to > application server connection**.

Note: If you are deploying a Java EE application, click the application menu next to the Java EE application in the Application Navigator.

For more information on creating application server connections, see [Section 32.2, "Creating a Connection to the Target Application Server"](#).

You may get an exception in JDeveloper when trying to deploy large EAR files. The workaround is to deploy the application using the Oracle WebLogic Server Administration Console.

32.8.2 How to Create an EAR File for Deployment

You can also use the deployment profile to create an archive file (EAR file). You can then deploy the archive file using the Oracle WebLogic Server Administration Console.

To create an archive file:

- In the Application Navigator, right-click the application containing the deployment profile, choose **Deploy > deployment profile > to EAR file**.

For Java EE applications, create an EAR deployment profile from the application context menu and assemble the projects, which typically represent Java EE modules, into your application archive.

Tip: Choose **View >Log** to see messages generated during creation of the archive file.

32.8.3 How to Deploy an Application Using Ant

You can deploy to most application servers from JDeveloper, or you can use tools provided by the application server vendor. You can also use Ant to package and deploy applications. The `build.xml` file, which contains the deployment commands for Ant, may vary depending on the target application server.

For deployment to other application servers, see the application server's documentation. If your application server does not provide specific Ant tasks, you may be able to use generic Ant tasks. For example, the generic `ear` task creates an EAR file for you.

For information about Ant, see <http://ant.apache.org>.

32.9 Testing the Application and Verifying Deployment

After you deploy the application, you can test it from Oracle WebLogic Server. To test-run your application, open a browser window and enter a URL:

- For non-Faces pages: `http://<host>:<port>/<context root>/<page>`
- For Faces pages: `http://<host>:<port>/<context root>/faces/<page>`

Note: /faces has to be in the URL for Faces pages. This is because JDeveloper configures your `web.xml` file to use the URL pattern of /faces in order to be associated with the Faces Servlet. The Faces Servlet does its per-request processing, strips out the /faces part in the URL, then forwards to the JSP. If you do not include the /faces in the URL, then the Faces Servlet is not engaged (since the URL pattern doesn't match). Your JSP is run without the necessary JSF per-request processing.

Part VI

Advanced Topics

Part VI contains the following chapters:

- [Chapter 33, "Advanced Business Components Techniques"](#)
- [Chapter 34, "Advanced Entity Object Techniques"](#)
- [Chapter 35, "Advanced View Object Techniques"](#)
- [Chapter 36, "Application State Management"](#)
- [Chapter 37, "Understanding Application Module Pooling"](#)

33

Advanced Business Components Techniques

This chapter describes advanced techniques that apply to all types of ADF Business Components.

This chapter includes the following sections:

- [Section 33.1, "Globally Extending ADF Business Components Functionality"](#)
- [Section 33.2, "Creating a Layer of Framework Extensions"](#)
- [Section 33.3, "Customizing Framework Behavior with Extension Classes"](#)
- [Section 33.4, "Creating Generic Extension Interfaces"](#)
- [Section 33.5, "Invoking Stored Procedures and Functions"](#)
- [Section 33.6, "Accessing the Current Database Transaction"](#)
- [Section 33.7, "Working with Libraries of Reusable Business Components"](#)
- [Section 33.8, "Customizing Business Components Error Messages"](#)
- [Section 33.9, "Creating Extended Components Using Inheritance"](#)
- [Section 33.10, "Substituting Extended Components In a Delivered Application"](#)

33.1 Globally Extending ADF Business Components Functionality

One of the powerful features of framework-based development is the ability to extend the base framework to change a built-in feature to behave differently or to add a new feature that can be used by all of your applications. This section describes:

- What framework extension classes are
- How to create an extension class and base ADF components you create on it
- How to adopt the best practice of using a whole custom layer of framework extension classes for your component or specific project

33.1.1 What Are ADF Business Components Framework Extension Classes?

An ADF Business Components framework extension class is Java class you write that extends one of the framework's base classes to:

- Augment a built-in feature works with additional, generic functionality
- Change how a built-in feature works, or even to
- Workaround a bug you encounter in a generic way

Once you've created a framework extension class, any new ADF components you create can be based on your *customized* framework class instead of the base one. Of course, you can also update the definitions of existing components to use the new framework extension class as well.

33.1.2 How To Create a Framework Extension Class

To create a framework extension class, follow these steps:

1. Identify a project to contain the framework extension class.

You can create it in the same project as your business service components if you believe it will only be used by components in that project. Alternatively, if you believe you might like to reuse the framework extension class across multiple Fusion web applications, create a separate `FrameworkExtensions` project to contain the framework extension classes.

2. Ensure the **BC4J Runtime** library is in the project's libraries list.

Use the **Libraries** page of the **Project Properties** dialog to verify this and to add the library if missing.

3. Create the new class using the **Create Java Class** dialog.

This dialog is available in the **New Gallery** in the **General** category.

4. Specify the appropriate framework base class from the `oracle.jbo.server` package in the **Extends** field.

[Figure 33–1](#) illustrates what it would look like to create a custom framework extension class named `CustomAppModuleImpl` in the `com.yourcompany.fwkext` package to customize the functionality of the base application module component. To quickly find the base class you're looking for, use the **Browse** button next to the **Extends** field that launches the JDeveloper **Class Browser**. Using its **Search** tab, you can type in part of the class name (including using * as a wildcard) to quickly subset the list of classes to find the one you're looking for.

Figure 33–1 Creating a Framework Extension Class for an Application Module



When you click **OK**, JDeveloper creates the custom framework extension class for you in the directory of the project's source path corresponding to the package name you've chosen.

Note: Some ADF Business Components component classes exist in both a server-side and a remote-client version. For example, if you use the JDeveloper **Class Browser** and type `ApplicationModuleImpl` into the **Match Class Name** field on the **Search** tab, the list will show two `ApplicationModuleImpl` classes: one in the `oracle.jbo.server` package and the other in the `oracle.jbo.client.remote` package. When creating framework extension classes, use the base ADF classes in the `oracle.jbo.server` package.

33.1.3 What Happens When You Create a Framework Extension Class

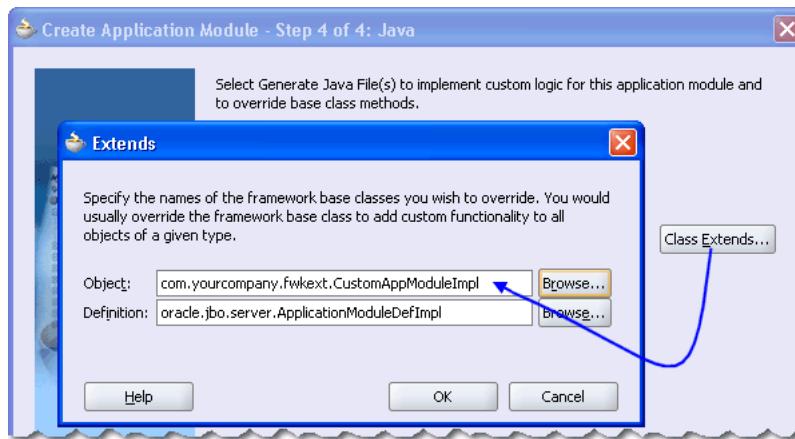
After creating a new framework extension class, it will not automatically be used by your application. You must decide which components in your project should make use of it. The following sections describe the available approaches for basing your ADF components on your own framework extension classes.

33.1.4 How to Base an ADF Component on a Framework Extension Class

You can set the base classes for any ADF component using the **Java** page of any ADF Business Components wizard or editor. Before doing so, review the following checklist:

- If you have decided to create your framework extension classes in a separate project, ensure that you have visited the **Dependencies** page of the **Project Properties** dialog for the project containing your business components in order to mark the **FrameworkExtension** project as a project dependency.
- If you have packaged your framework extension classes in a Java archive (JAR) file, ensure that you have created a named library definition to reference its JAR file and also listed that library in the library list of the project containing your business components. To create a library if missing, use the **Manage Libraries** dialog available from the **Tools | Manage Libraries** main menu item. To verify or adjust the project's library list, use the **Libraries** page of the **Project Properties** dialog.

After you ensure the framework classes are available to reference, [Figure 33–2](#) shows how you would use the `CustomApplicationModuleImpl` class as the base class for a new application module. By clicking the **Class Extends** button of the **Java** page of the wizard, the **Extends** dialog displays to let you enter the fully-qualified name of your framework extension class (or use the **Browse** button to use the JDeveloper **Class Browser** to find it quickly).

Figure 33–2 Specifying a Custom Base Class for a New Application Module

The same **Class Extends** button appears on the **Java** page of every ADF Business Components wizard and editor, so you can use this technique to choose your desired framework extension base class(es) both for new components or existing ones.

Note: When using the JDeveloper **Class Browser** in the **Extends** dialog of an ADF Business Components wizard or editor to select a custom base class for the component, the list of available classes is automatically filtered to show only classes that are appropriate. For example, when clicking **Browse** in Figure 33–2 to select an application module **Object** base class, the list will only show classes available in the current project's library list which extend the `oracle.jbo.server.ApplicationModule` class either directly or indirectly. If you don't see the class you're looking for, either you extended the incorrect base class or you have chosen the wrong component class name to override.

33.1.5 What Happens When You Base a Component on a Framework Extension Class

When an ADF component you create extends a custom framework extension class, JDeveloper updates its XML component definition to reflect the custom class name you've chosen.

33.1.5.1 Basing an XML-Only Component on a Framework Extension Class

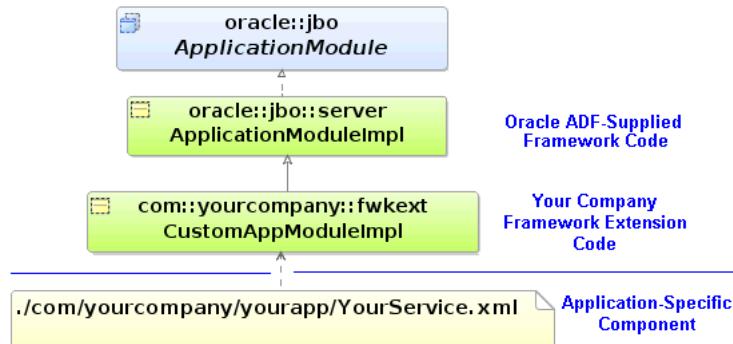
For example, assume you've created the `YourService` application module above in the `com.yourcompany.yourapp` package, with a custom application module base class of `CustomAppModuleImpl`. If you have opted to leave the component as an XML-only component with no custom Java file, its XML component definition (`YourService.xml`) will look like what you see in Example 33–1. The value of the `ComponentClass` attribute of the `< AppModule >` tag is read at runtime to identify the Java class to use to represent the component.

Example 33–1 Custom Base Class Names Are Recorded in XML Component Definition

```
< AppModule
    Name="YourService"
    ComponentClass="com.yourcompany.fwkext.CustomAppModuleImpl" >
    <!-- etc. -->
</ AppModule >
```

Figure 33–3 illustrates how the XML-only YourService application module relates to your custom extension class. At runtime, it uses the CustomAppModuleImpl class which inherits its base behavior from the ApplicationModuleImpl class.

Figure 33–3 XML-Only Component Reference an Extended Framework Base Class



33.1.5.2 Basing a Component with a Custom Java Class on a Framework Extension Class

If your component requires a custom Java class, as you've seen in previous chapters you open the **Java** page of the component editor and check the appropriate checkbox to enable it. For example, when you enable a custom application module class for the YourServer application module, JDeveloper creates the appropriate YourServiceImpl.java class. As shown in [Example 33–2](#), it also updates the component's XML component definition to reflect the name of the custom component class.

Example 33–2 Custom Component Class Recorded in XML Component Definition

```

< AppModule
    Name="YourService"
    ComponentClass="com.yourcompany.yourapp.YourServiceImpl" >
    <!-- etc. -->
</ AppModule>

```

JDeveloper also updates the component's custom Java class to modify its extends clause to reflect the new custom framework base class, as shown in [Example 33–3](#).

Example 33–3 Component's Custom Java Class Updates to Reflect New Base Class

```

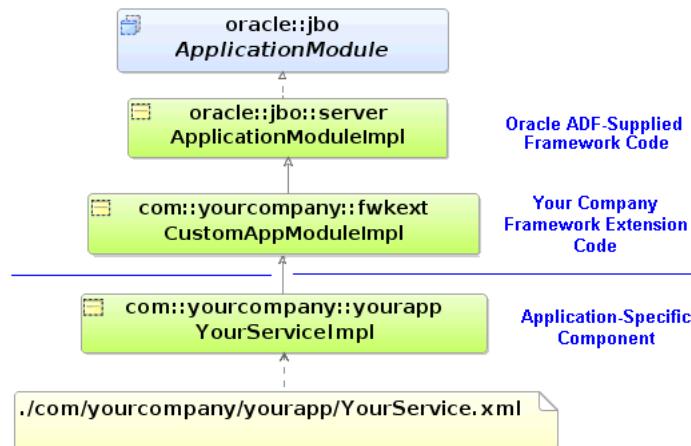
package com.yourcompany.yourapp;
import com.yourcompany.fwkext.CustomAppModuleImpl;
// -----
// --- File generated by Oracle ADF Business Components Design Time.
// --- Custom code may be added to this class.
// --- Warning: Do not modify method signatures of generated methods.
// -----
public class YourServiceImpl extends CustomAppModuleImpl {
    /**This is the default constructor (do not remove) */
    public YourServiceImpl() {}
    // etc.
}

```

Figure 33–4 illustrates how the YourService application module with its custom YourServiceImpl class is related to your framework extension class. At runtime, it

uses the `YourServiceImpl` class which inherits its base behavior from the `Custom AppModuleImpl` framework extension class which, in turn, extends the base `ApplicationModuleImpl` class.

Figure 33–4 Component with Custom Java Extending Customized Framework Base Class



33.1.6 What You May Need to Know

33.1.6.1 Don't Update the Extends Clause in Custom Component Java Files By Hand

If you have an ADF component with a custom Java class and later decide to base the component on a framework extension class, use the **Class Extends** button on the **Java** page of the component editor to change the component's base class. Doing this updates the component's XML component definition to reflect the new base class, and *also* modifies the `extends` clause in the component's custom Java class. If you manually update the `extends` clause without using the component editor, the component's XML component definition will not reflect the new inheritance and the next time you open the editor, your manually modified `extends` clause will be overwritten with what the component editor believes is the correct component base class.

33.1.6.2 You Can Have Multiple Levels of Framework Extension Classes

In the examples above, you've seen a single `Custom AppModuleImpl` class that extends the base `ApplicationModuleImpl` class. However, there is no fixed limit on how many levels of framework extension classes you create. After creating a company-level `Custom AppModuleImpl` to use for all application modules in all Fusion web applications your company creates, some later project team might encounter the need to further customize that framework extension class. That team could create a `SomeProjectCustom AppModuleImpl` class that extends the `Custom AppModuleImpl` and then include the project-specific custom application module code in there:

```

public class SomeProjectCustom AppModuleImpl
    extends Custom AppModuleImpl {
    /*
     * Custom application module code specific to the
     * "SomeProject" project goes here.

```

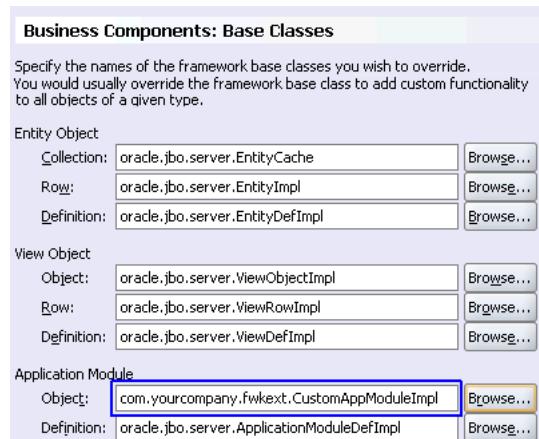
```
* /
}
```

Then, any application modules created as part of the implementation of this specific project can use the `SomeProjectCustomAppModuleImpl` as their base class instead of the `CustomAppModuleImpl`.

33.1.6.3 Setting up Project-Level Preferences for Framework Extension Classes

If you decide to use a specific set of framework extension classes as a standard for a given project, you can open the **Business Components > Base Classes** page in the **Project Properties** dialog, as shown in [Figure 33–5](#), to define your preferred base classes for each component type. For example, to indicate that any new application modules created in the project should use the `CustomAppModuleImpl` class by default, enter the fully-qualified name of that class in the **Application Module Object** class name field as shown. Setting these preferences for base classes does not affect any existing components in the project, but the component wizards will use the preferences for any new components created.

Figure 33–5 Setting Project-Level Preferences for ADF Component Base Classes



33.1.6.4 Setting Up Framework Extension Class Preferences at the IDE Level

When you want to apply the same base class preferences to each new project that you create in JDeveloper, you can define the preferences at a global level. Choose the **Tools | Preferences** main menu item in JDeveloper and display the **Business Components > Base Classes** page. The page displays the same options for specifying the preferred base classes for each component type as shown in [Figure 33–5](#). Base classes that you specify at the global level will not alter your existing projects containing ADF components.

33.2 Creating a Layer of Framework Extensions

Before you begin to develop application-specific business components, Oracle recommends that you consider creating a complete layer of framework extension classes and setting up your project-level preferences to use that layer by default. You might not have any custom code in mind to put in these framework extension classes yet, but you will be glad you heeded this recommendation the first time you encounter a need to:

- Add a generic feature that all your company's application modules require

- Augment a built-in feature with some custom, generic processing
- Workaround a bug you encounter in a generic way

Failure to set up these preferences at the outset can present your team with a substantial inconvenience if you discover mid-project that all of your entity objects, for example, require a new generic feature, augmented built-in feature, or a generic bug workaround. Putting a complete layer of framework classes in place at the start of your project is an insurance policy against this inconvenience and the wasted time related to dealing with it later in the project. JDeveloper will automatically use framework classes when you create them.

33.2.1 How to Create Your Layer of Framework Extension Layer Classes

A common set of customized framework base classes in a package name of your own choosing like `com.yourcompany.adfextensions`, each importing the `oracle.jbo.server.*` package, would consist of the following classes:

- `public class CustomEntityImpl extends EntityImpl`
- `public class CustomEntityDefImpl extends EntityDefImpl`
- `public class CustomViewObjectImpl extends ViewObjectImpl`
- `public class CustomViewRowImpl extends ViewRowImpl`
- `public class CustomApplicationModuleImpl extends ApplicationModuleImpl`
- `public class CustomDBTransactionImpl extends DBTransactionImpl2`
- `public class CustomDatabaseTransactionFactory extends DatabaseTransactionFactory`

To make your framework extension layer classes easier to package as a reusable library, Oracle recommends creating them in a separate project from the projects that use them. For details about using the custom `DBTransactionImpl` class, see [Section 33.8.4.2, "Configuring an Application Module to Use a Custom Database Transaction Class"](#).

Note: For your convenience, the `FrameworkExtensions` project in the `AdvancedExamples` workspace contains a set of these classes. You can select the `com.yourcompany.adfextensions` package in the Application Navigator and choose the **Refactor > Rename** option from the context menu to change the package name of all the classes to a name you prefer.

For completeness, you may also want to create customized framework classes for the following classes as well, note however that overriding anything in these classes would be a fairly rare requirement.

- `public class CustomViewDefImpl extends ViewDefImpl`
- `public class CustomEntityCache extends EntityCache`
- `public class CustomApplicationModuleDefImpl extends ApplicationModuleDefImpl`

33.2.2 How to Package Your Framework Extension Layer in a JAR File

Use the **Create Deployment Profile: JAR File** dialog to create a JAR file containing the classes in your framework extension layer. This is available in the **New Gallery** in the **General > Deployment Files** category.

Give the deployment profile a name like `FrameworkExtensions` and click **OK**. By default the JAR file will include all class files in the project. Since this is exactly what you want, when the **JAR Deployment Profile Properties** dialog appears, you can just click **OK** to finish.

Note: Do not use the **ADF Library JAR** archive type to package your framework extension layer. You create the ADF Library JAR file when you want to package reusable components to share in the JDeveloper Resource Catalog. For details about working with ADF components and the ADF Library JAR archive type, see [31.2 , "Packaging a Reusable ADF Component into an ADF Library"](#).

Finally, to create the JAR file, right-click the project folder in the Application Navigator and choose **Deploy - *YourProfileName* - to JAR File** on the context menu. A **Deployment** tab appears in the JDeveloper **Log window** that should display feedback like:

```
---- Deployment started. ---- Feb 14, 2008 1:42:39 PM
Running dependency analysis...
Wrote JAR file to ...\\FrameworkExtensions\\deploy\\FrameworkExtensions.jar
Elapsed time for deployment: 2 seconds
---- Deployment finished. ---- Reb 14, 2008 1:42:41 PM
```

33.2.3 How to Create a Library Definition for Your Framework Extension JAR File

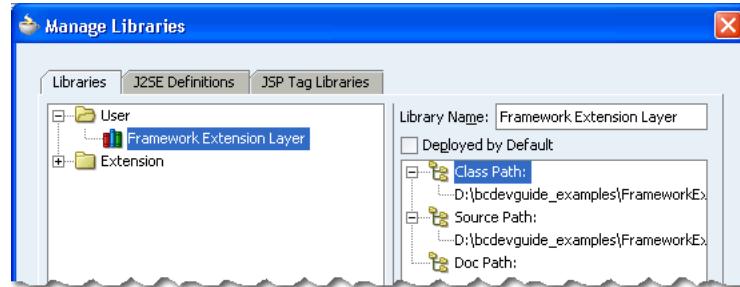
JDeveloper uses named libraries as a convenient way to organize the one or more JAR files that comprise reusable component libraries. To define a library for your framework extensions JAR file, do the following:

1. Choose **Tools > Manage Libraries** from the JDeveloper main menu.
2. In the **Manage Libraries** dialog, select the **Libraries** tab.
3. Select the **User** folder in the tree and click the **New** button.
4. In the **Create Library** dialog that appears, name the library "Framework Extension Layer" and select the **Class Path** node and click **Add Entry**.
5. Use the **Select Path Entry** dialog that appears to select the `FrameworkExtensions.jar` file that contains the class files for the framework extension components, then click **Select**.
6. Select the **Source Path** node and click **Add Entry**.
7. Use the **Select Path Entry** dialog that appears to select the `..\\FrameworkExtensions\\src` directory where the source files for the framework extension classes reside, then click **Select**.
8. Click **OK** to dismiss the **Create Library** dialog and define the new library.

When finished, you will see your new "Framework Extension Layer" user-defined library, as shown in [Figure 33-6](#). You can then add this library to the library list of any project where you will be building business services, and your custom framework

extension classes will be available to reference as the preferred component base classes.

Figure 33–6 New User-Defined Library for Your Framework Extensions Layer

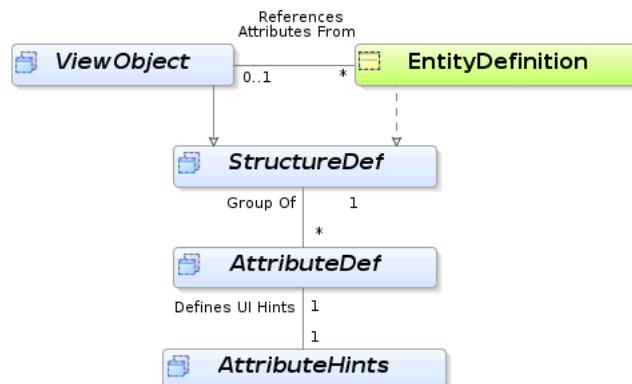


33.3 Customizing Framework Behavior with Extension Classes

One of the common tasks you'll perform in your framework extension classes is implementing custom application functionality. Since framework extension code is written to be used by all components of a specific type, the code you write in these classes often needs to work with component attributes in a generic way. To address this need, ADF provides API's that allow you to access component metadata at runtime. It also provides the ability to associate custom metadata properties with any component or attribute. You can write your generic framework extension code to leverage runtime metadata and custom properties to build generic functionality, which if necessary, only is used in the presence of certain custom properties.

33.3.1 How to Access Runtime Metadata For View Objects and Entity Objects

Figure 33–7 illustrates the three primary interfaces ADF provides for accessing runtime metadata about view objects and entity objects. The `ViewObject` interface extends the `StructureDef` interface. The class representing the entity definition (`EntityDefImpl`) also implements this interface. As its name implies, the `StructureDef` defines the structure and the component and provides access to a collection of `AttributeDef` objects that offer runtime metadata about each attribute in the view object row or entity row. Using an `AttributeDef`, you can access its companion `AttributeHints` object to reference hints like the display label, format mask, tooltip, etc.

Figure 33–7 Runtime Metadata Available for View Objects and Entity Objects

33.3.2 Implementing Generic Functionality Using Runtime Metadata

In [Section 6.4.1, "ViewObject Interface Methods for Working with the View Object's Default RowSet"](#) you learned that for read-only view objects the `findByPrimaryKey()` method and the `setCurrentRowWithKey` built-in operation only work if you override the `create()` method on the view object to call `setManageRowsByKey(true)`. This can be a tedious detail to remember if you create a lot of read-only view objects, so it is a great candidate for automating in a framework extension class for view objects.

Assume a `FrameworkExtensions` project contains a `FODViewObjectImpl` class that is the base class for all view objects in the application. This framework extension class for view objects extends the base `ViewObjectImpl` class and overrides the `create()` method as shown in [Example 33–4](#) to automate this task. After calling the `super.create()` to perform the default framework functionality when a view object instance is created at runtime, the code tests whether the view object is a read-only view object with at least one attribute marked as a key attribute. If this is the case, it invokes `setManageRowsByKey(true)`.

The `isReadOnlyNonEntitySQLViewWithAtLeastOneKeyAttribute()` helper method determines whether the view object is read-only by testing the combination of the following conditions:

- `isFullSql()` is true

This method returns true if the view object's SQL query is completely specified by the developer, as opposed to having the select list derived automatically based on the participating entity usages.

- `getEntityDefs()` is null

This method returns an array of `EntityDefImpl` objects representing the view object's entity usages. If it returns null, then the view object has no entity usages.

It goes on to determine whether the view object has any key attributes by looping over the `AttributeDef` array returned by the `getAttributeDefs()` method. If the `isPrimaryKey()` method returns true for any attribute definition in the list, then you know the view object has a key.

Example 33–4 Automating Setting Manage Rows By Key

```

public class FODViewObjectImpl extends ViewObjectImpl {
    protected void create() {
        super.create();
        if (isReadOnlyNonEntitySQLViewWithAtLeastOneKeyAttribute()) {
    
```

```

        setManageRowsByKey(true);
    }
}
boolean isReadOnlyNonEntitySQLViewWithAtLeastOneKeyAttribute() {
    if (getViewDef().isFullSql() && getEntityDefs() == null) {
        for (AttributeDef attrDef : getAttributeDefs()) {
            if (attrDef.isPrimaryKey()) {
                return true;
            }
        }
    }
    return false;
}
// etc.
}

```

33.3.3 Implementing Generic Functionality Driven by Custom Properties

In JDeveloper, when you create application modules, view objects, and entity objects you can select the **General** navigation tab in these business component's overview editor and expand the **Custom Properties** section to define custom metadata properties for any component. These are name/value pairs that you can use to communicate additional declarative information about the component to the generic code that you write in framework extension classes. You can use the `getProperty()` method in your code to conditionalize generic functionality based on the presence of, or the specific value of, one of these custom metadata properties.

For example, the `FODViewObjectImpl` framework extension class overrides the view object's `insertRow()` method as shown in [Example 33–5](#) to conditionally force a row to be inserted and to appear as the last row in the row set. If any view object extending this framework extension class defines a custom metadata property named `InsertNewRowsAtEnd`, then this generic code executes to insert new rows at the end. If a view object does not define this property, it will have the default `insertRow()` behavior.

Example 33–5 Conditionally Inserting New Rows at the End of a View Object's Default RowSet

```

public class FODViewObjectImpl extends ViewObjectImpl {
    private static final String INSERT_NEW_ROWS_AT_END = "InsertNewRowsAtEnd";
    public void insertRow(Row row) {
        super.insertRow(row);
        if (getProperty(INSERT_NEW_ROWS_AT_END) != null) {
            row.removeAndRetain();
            last();
            next();
            getDefaultRowSet().insertRow(row);
        }
    }
    // etc.
}

```

In addition to defining component-level custom properties, you can also define properties on view object attributes, entity object attributes, and domains. At runtime, you access them using the `getProperty()` method on the `AttributeDef` interface for a given attribute.

33.3.4 What You May Need to Know

33.3.4.1 Determining the Attribute Kind at Runtime

In addition to providing information about an attribute's name, Java type, SQL type, and many other useful pieces of information, the `AttributeDef` interface contains the `getAttributeKind()` method that you can use to determine the kind of attribute it represents. This method returns a byte value corresponding to one of the public constants in the `AttributeDef` interface listed in [Table 33-1](#).

Table 33-1 Entity Object and View Object Attribute Kinds

Public AttributeDef Constant	Attribute Kind Description
<code>ATTR_PERSISTENT</code>	Persistent attribute
<code>ATTR_TRANSIENT</code>	Transient attribute
<code>ATTR_ENTITY_DERIVED</code>	View object attribute mapped to an entity-level transient attribute
<code>ATTR_SQL_DERIVED</code>	SQL-Calculated attribute
<code>ATTR_DYNAMIC</code>	Dynamic attribute
<code>ATTR_ASSOCIATED_ROWITERATOR</code>	Accessor attribute returning a RowSet of set of zero or more Rows
<code>ATTR_ASSOCIATED_ROW</code>	Accessor attribute returning a single Row

33.3.4.2 Configuring Design Time Custom Property Names

Once you have written framework extension classes that depend on custom properties, you can set a JDeveloper preference so that your custom property names show in the list on the **Custom Properties** section of the corresponding component editor. To set up these pre-defined custom property names, choose **Tools | Preferences** from the JDeveloper main menu and open the **Business Components > Property Names** tab in the **Preferences** dialog.

33.3.4.3 Setting Custom Properties at Runtime

You may find it handy to programmatically set custom property values at runtime. While the `setProperty()` API to perform this function is by design not available to clients on the `ViewObject`, `ApplicationModule`, or `AttributeDef` interfaces in the `oracle.jbo` package, code you write *inside* your ADF components' custom Java classes can use it.

33.4 Creating Generic Extension Interfaces

In addition to creating framework extension classes, you can create custom interfaces that all of your components can implement by default. The client interface is very useful for exposing methods from your application module that might be invoked by UI clients, for example. This section considers an example for an application module, however, the same functionality is possible for a custom extended view object and view row interface as well. For more information about client interfaces, see also [Section 9.8, "Publishing Custom Service Methods to UI Clients"](#) and [Section 9.10, "Working Programmatically with an Application Module's Client Interface"](#).

Assume that you have a `CustomApplicationModuleImpl` class that extends `ApplicationModuleImpl` and that you want to expose two custom methods like this:

```
public void doFeatureOne(String arg);
public int anotherFeature(String arg);
```

Perform the following steps to create a custom extension interface `CustomApplicationModule` and have your `CustomApplicationModuleImpl` class implement it.

1. Create a custom interface that contains the methods you would like to expose globally on your application module components. For this scenario, that interface would look like this:

```
package devguide.advanced.customintf.fwkext;
/**
 * NOTE: This does not extend the
 * ===== oracle.jbo.ApplicationModule interface.
 */
public interface CustomApplicationModule {
    public void doFeatureOne(String arg);
    public int anotherFeature(String arg);
}
```

Notice that the interface does *not* extend the `oracle.jbo.ApplicationModule` interface.

2. Modify your `CustomApplicationModuleImpl` application module framework extension class to implement this new `CustomApplicationModule` interface.

```
package devguide.advanced.customintf.fwkext;
import oracle.jbo.server.ApplicationModuleImpl;
public class CustomApplicationModuleImpl
    extends ApplicationModuleImpl
    implements CustomApplicationModule {
    public void doFeatureOne(String arg) {
        System.out.println(arg);
    }
    public int anotherFeature(String arg) {
        return arg == null ? 0 : arg.length();
    }
}
```

3. Rebuild your project.

The ADF wizards will only "see" your interfaces after they have been successfully compiled.

After you have implemented your `CustomApplicationModuleImpl` class, you can create a new application module which exposes the global extension interface and is based on your custom framework extension class. For this purpose you use the application module editor.

To create a custom application module interface:

1. In the Application Navigator, double-click the application module for which you want to create the custom interface.

For example, you might create a new `ProductModule` application module which exposes the global extension interface `CustomApplicationModule` and is based on the `CustomApplicationModuleImpl` framework extension class.

2. In the overview editor navigation list, select **Java**. The Java Classes page should show an existing Java class for the application module identified as **Application Module Class**.

By default, JDeveloper generates the Java class for application modules you create. However, if you disabled this feature, click the **Edit java options** button in the Java Classes section and select **Generate Application Module Class**. Click **OK** to add a Java class to the project from which you will create the custom interface.

3. In the Java Classes page of the overview editor, click the **Edit Java option** button and click **Class Extends**. In the Override Base Classes dialog, specify the name of the framework base class you want to override and click **OK**.

For example, you might select `CustomApplicationModuleImpl` as the base class for the application module.

4. In the overview editor, expand the **Client Interface** section of the Java Classes page and click the **Edit application module client interface** button. In the Edit Client Interface dialog, click the **Interfaces** button.
5. In the Select Interfaces to Extend dialog, select the desired custom application module interface from the available list. Click **OK** to return to the Edit Client Interfaces dialog.

For example, you might shuttle the `CustomApplicationModule` interface to the **Selected** list to be one of the custom interfaces that clients can use with your component.

6. In the Edit Client Interfaces dialog, ensure that at least one method appears in the **Selected** list. Click **OK** to return to the application module editor. The Java Classes page displays the new custom interface for the application module identified as **Application Module Client Interface**.

Note: You need to select at least one method in the **Selected** list in the Edit Client Interfaces dialog, even if it means redundantly selecting one of the methods on the global extension interface. Any method will do in order to get JDeveloper to generate the custom interface.

When you dismiss the Edit Client Interfaces dialog and return to the application module editor, JDeveloper generates the application module custom interface. For example, the custom interface `ProductModule` automatically extends *both* the base `ApplicationModule` interface and your `CustomApplicationModule` extension interface like this:

```
package devguide.advanced.customintf.common;
import devguide.advanced.customintf.fwkext.CustomApplicationModule;

import oracle.jbo.ApplicationModule;
// -----
// --- File generated by Oracle ADF Business Components Design Time.
// -----
public interface ProductModule
    extends CustomApplicationModule, ApplicationModule {
    void doSomethingProductRelated();
}
```

Once you've done this, then client code can cast your `ProductModule` application module to a `CustomApplicationModule` interface and invoke the generic extension methods it contains in a strongly-typed way.

Note: The basic steps are the same for exposing methods on a `ViewObjectImpl` framework extension class, as well as for a `ViewRowImpl` extension class.

33.5 Invoking Stored Procedures and Functions

You can write code in the custom Java classes for your business components to invoke database stored procedures and functions. Here you'll consider some simple examples based on procedures and functions in a PL/SQL package; however, using the same techniques, you also can invoke procedures and functions that are not part of a package.

Consider the following PL/SQL package:

```
create or replace package devguidepkg as
    procedure proc_with_no_args;
    procedure proc_with_three_args(n number, d date, v varchar2);
    function func_with_three_args(n number, d date, v varchar2) return varchar2;
    procedure proc_with_out_args(n number, d out date, v in out varchar2);
end devguidepkg;
```

The following sections explain how to invoke each of the example procedures and functions in this package.

33.5.1 Invoking Stored Procedures with No Arguments

If you need to invoke a stored procedure that takes no arguments, you can use the `executeCommand()` method on the `DBTransaction` interface (in the `oracle.jbo.server` package as shown in [Example 33–6](#).

Example 33–6 Executing a Stored Procedure with No Arguments

```
// In StoredProcedureTestModuleImpl.java
public void callProcWithNoArgs() {
    getDBTransaction().executeCommand(
        "begin devguidepkg.proc_with_no_args; end; ");
}
```

33.5.2 Invoking Stored Procedure with Only IN Arguments

Invoking stored procedures that accept only `IN`-mode arguments — which is the default PL/SQL parameter mode if not specified — requires using a JDBC `PreparedStatement` object. The `DBTransaction` interface provides a `createPreparedStatement()` method to create this object for you in the context of the current database connection. You could use a helper method like the one shown in [Example 33–7](#) to simplify the job of invoking a stored procedure of this kind using a `PreparedStatement`. Importantly, by using a helper method, you can encapsulate the code that closes the JDBC `PreparedStatement` after executing it. The code performs the following basic tasks:

1. Creates a JDBC `PreparedStatement` for the statement passed in, wrapping it in a PL/SQL `begin...end` block.
2. Loops over values for the bind variables passed in, if any.

3. Sets the value of each bind variable in the statement.

Notice that since JDBC bind variable API's use one-based numbering, the code adds one to the zero-based for loop index variable to account for this.

4. Executes the statement.
5. Closes the statement.

Example 33–7 Helper Method to Simplify Invoking Stored Procedures with Only IN Arguments

```
protected void call.StoredProcedure(String stmt, Object[] bindVars) {
    PreparedStatement st = null;
    try {
        // 1. Create a JDBC PreparedStatement for
        st = getDBTransaction().createPreparedStatement("begin "+stmt+";end;", 0);
        if (bindVars != null) {
            // 2. Loop over values for the bind variables passed in, if any
            for (int z = 0; z < bindVars.length; z++) {
                // 3. Set the value of each bind variable in the statement
                st.setObject(z + 1, bindVars[z]);
            }
        }
        // 4. Execute the statement
        st.executeUpdate();
    }
    catch (SQLException e) {
        throw new JboException(e);
    }
    finally {
        if (st != null) {
            try {
                // 5. Close the statement
                st.close();
            }
            catch (SQLException e) {}
        }
    }
}
```

With a helper method like this in place, calling the `proc_with_three_args` procedure above would look like this:

```
// In StoredProcedureTestModuleImpl.java
public void callProcWithThreeArgs(Number n, Date d, String v) {
    call.StoredProcedure("devguidepkg.proc_with_three_args(?, ?, ?)",
        new Object[]{n, d, v});
}
```

Notice the question marks used as JDBC bind variable placeholders for the arguments passed to the function. JDBC also supports using named bind variables, but using these simpler positional bind variables is also fine since the helper method is just setting the bind variable values positionally.

33.5.3 Invoking Stored Function with Only IN Arguments

Invoking stored functions that accept only `IN`-mode arguments requires using a JDBC `CallableStatement` object in order to access the value of the function result after executing the statement. The `DBTransaction` interface provides a `createCallableStatement()` method to create this object for you in the context of

the current database connection. You could use a helper method like the one shown in [Example 33–8](#) to simplify the job of invoking a stored function of this kind using a CallableStatement. As above, the helper method encapsulates both the creation and clean up of the JDBC statement being used.

The code performs the following basic tasks:

1. Creates a JDBC CallableStatement for the statement passed in, wrapping it in a PL/SQL begin...end block.
2. Registers the first bind variable for the function return value.
3. Loops over values for the bind variables passed in, if any.
4. Sets the value of each bind user-supplied bind variable in the statement.

Notice that since JDBC bind variable API's use one-based numbering, and since the function return value is already the first bind variable in the statement, the code adds *two* to the zero-based for loop index variable to account for these.

5. Executes the statement.
6. Returns the value of the first bind variable.
7. Closes the statement.

Example 33–8 Helper Method to Simplify Invoking Stored Functions with Only IN Arguments

```
// Some constants
public static int NUMBER = Types.NUMERIC;
public static int DATE = Types.DATE;
public static int VARCHAR2 = Types.VARCHAR;

protected Object callStoredFunction(int sqlReturnType, String stmt,
                                    Object[] bindVars) {
    CallableStatement st = null;
    try {
        // 1. Create a JDBC CallableStatement
        st = getDBTransaction().createCallableStatement(
            "begin ? := "+stmt+";end;",0);
        // 2. Register the first bind variable for the return value
        st.registerOutParameter(1, sqlReturnType);
        if (bindVars != null) {
            // 3. Loop over values for the bind variables passed in, if any
            for (int z = 0; z < bindVars.length; z++) {
                // 4. Set the value of user-supplied bind vars in the stmt
                st.setObject(z + 2, bindVars[z]);
            }
        }
        // 5. Set the value of user-supplied bind vars in the stmt
        st.executeUpdate();
        // 6. Return the value of the first bind variable
        return st.getObject(1);
    }
    catch (SQLException e) {
        throw new JboException(e);
    }
    finally {
        if (st != null) {
            try {
                // 7. Close the statement
                st.close();
            }
            catch (SQLException e) {
                throw new JboException(e);
            }
        }
    }
}
```

```
        }  
    catch (SQLException e) {}  
}
```

With a helper method like this in place, calling the `func_with_three_args` procedure above would look like this:

```
// In StoredProcTestModuleImpl.java
public String callFuncWithThreeArgs(Number n, Date d, String v) {
    return (String)callStoredFunction(VARCHAR2,
        "devguidepkg.func_with_three_args(?, ?, ?)",
        new Object[]{n,d,v});
}
```

Notice the question marks as above that are used as JDBC bind variable placeholders for the arguments passed to the function. JDBC also supports using named bind variables, but using these simpler positional bind variables is also fine since the helper method is just setting the bind variable values positionally.

33.5.4 Calling Other Types of Stored Procedures

Calling a stored procedure or function like `devguidepkg.proc_with_out_args` that includes arguments of `OUT` or `IN OUT` mode requires using a `CallableStatement` as in the previous section, but is a little more challenging to generalize into a helper method. [Example 33-9](#) illustrates the JDBC code necessary to invoke the `devguidepkg.proc_with_out_args` procedure.

The code performs the following basic tasks:

1. Defines a PL/SQL block for the statement to invoke.
 2. Creates the CallableStatement for the PL/SQL block.
 3. Registers the positions and types of the OUT parameters.
 4. Sets the bind values of the IN parameters.
 5. Executes the statement.
 6. Creates a JavaBean to hold the multiple return values
The DateAndStringBean class contains bean properties
stringVal.
 7. Sets the value of its dateVal property using the first OUT p
 8. Sets value of its stringVal property using second OUT p
 9. Returns the result.
 10. Closes the JDBC CallableStatement.

Example 33–9 Calling a Stored Procedure with Multiple OUT Arguments

```
public Date callProcWithOutArgs(Number n, String v) {  
    CallableStatement st = null;  
    try {  
        // 1. Define the PL/SQL block for the statement to invoke  
        String stmt = "begin devguidepkg.proc_with_out_args(?, ?, ?); end;";  
        // 2. Create the CallableStatement for the PL/SQL block  
        st = getDBTransaction().createCallableStatement(stmt, 0);  
        // 3. Register the positions and types of the OUT parameters
```

```
        st.registerOutParameter(2,Types.DATE);
        st.registerOutParameter(3,Types.VARCHAR);
        // 4. Set the bind values of the IN parameters
        st.setObject(1,n);
        st.setObject(3,v);
        // 5. Execute the statement
        st.executeUpdate();
        // 6. Create a bean to hold the multiple return values
        DateAndStringBean result = new DateAndStringBean();
        // 7. Set value of dateValue property using first OUT param
        result.setDateVal(new Date(st.getDate(2)));
        // 8. Set value of stringValue property using 2nd OUT param
        result.setStringVal(st.getString(3));
        // 9. Return the result
        return result;
    } catch (SQLException e) {
        throw new JboException(e);
    } finally {
        if (st != null) {
            try {
                // 10. Close the JDBC CallableStatement
                st.close();
            }
            catch (SQLException e) {}
        }
    }
}
```

The DateAndString bean used in [Example 33–9](#) is a simple JavaBean with two bean properties like this:

```
package devguide.advanced.storedproc;
import java.io.Serializable;
import oracle.jbo.domain.Date;
public class DateAndStringBean implements Serializable {
    Date dateVal;
    String stringVal;
    public void setDateVal(Date dateVal) {this.dateVal=dateVal;}
    public Date getDateVal() {return dateVal;}
    public void setStringVal(String stringVal) {this.stringVal=stringVal;}
    public String getStringVal() {return stringVal;}
}
```

Note: In order to allow the custom method to be a legal candidate for inclusion in an application module's custom service interface (if desired), the bean needs to implement the `java.io.Serializable` interface. Since this is a "marker" interface, this involves simply adding the `implements Serializable` keywords without needing to code the implementation of any interface methods.

33.6 Accessing the Current Database Transaction

Since the ADF Business Components components abstract all of the lower-level database programming details for you, you typically won't need *direct* access to the JDBC Connection object. Unless you use the reserved release mode described in [Section 36.2.2.3.3, "About Reserved Release Level"](#), there is no guarantee at runtime that your application will use the exact same application module instance or JDBC

Connection instance across different web page requests. Since inadvertently holding a reference to the JDBC Connection object in this type of pooled services environment can cause unpredictable behavior at runtime, by design, the ADF Business Components layer has no direct API to obtain the JDBC Connection. This is an intentional attempt to discourage its direct use and inadvertent abuse.

However, on occasion it may come in handy when you're trying to integrate third-party code with ADF Business Components, so you can use a helper method like the one shown in [Example 33–10](#) to access the connection.

Example 33–10 Helper Method to Access the Current JDBC Connection

```
/**  
 * Put this method in your XXXXImpl.java class where you need  
 * to access the current JDBC connection  
 */  
private Connection getCurrentConnection() throws SQLException {  
    /* Note that we never execute this statement, so no commit really happens */  
    PreparedStatement st = getDBTransaction().createPreparedStatement("commit",1);  
    Connection conn = st.getConnection();  
    st.close();  
    return conn;  
}
```

Caution: Oracle recommends that you never cache the JDBC connection obtained using the helper method above in your own code anywhere. Instead, call the helper method each time you need it to avoid inadvertently holding a reference to a JDBC Connection that might be used in another request by another user at a later time do to the pooled services nature of the ADF runtime environment.

33.7 Working with Libraries of Reusable Business Components

As with other Java components, you can create a JAR file containing one or more packages of reusable ADF components. Then, in other projects you can import one or more packages of components from this component library to reference those in a new application.

33.7.1 How To Create a Reusable Library of Business Components

Use the **Create Business Components Archive Profile** dialog to create a JAR file containing the Java classes and XML component definitions that comprise your business components library. This is available in the **New Gallery** in the **General > Deployment Profiles** category.

Note: If you don't see the **Deployment Profiles** category in the **New Gallery**, set the **Filter By** dropdown list at the top of the dialog to the **All Technologies** choice to make it visible.

Give the deployment profile a name like `ReusableComponents` and click **OK**. As the Project Properties dialog shows in [Figure 33–8](#), the `ReusableComponents` business components deployment archive profile contains two nested JAR deployment profiles:

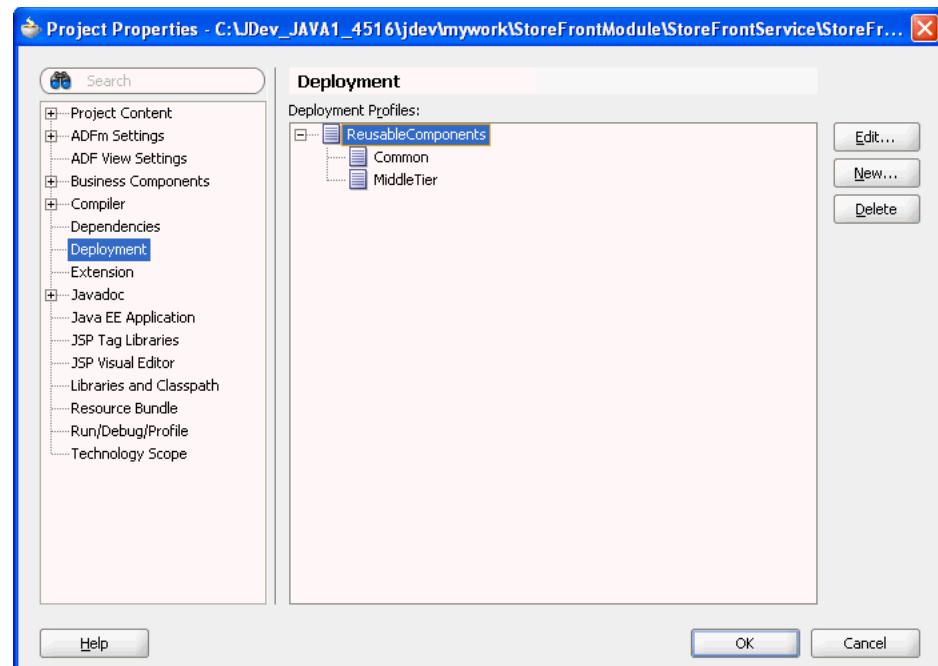
- `Common.deploy`
- `MiddleTier.deploy`

These two nested profiles are standard JAR deployment profiles that are pre-configured to bundle:

- All of the business components custom java classes and XML component definitions into a `ReusableComponentsCSMT.jar` archive
- All of the client interfaces, message bundle classes, and custom domains into a `ReusableComponentsCSCommon.jar`

They are partitioned this way in order to simplify deployment of ADF Business Components-based applications. The `*CSMT.jar` is an archive of components designed to be deployed only on the middle tier application server. The `*CSCommon.jar` is common both to the application server and to the remote client tier in the deployment scenario when the client interacting with the application module is running in a different physical server from the application module with which it is working.

Figure 33–8 Business Components Archive Deployment Profile Contains Nested Profiles



To create the JAR files, in the Application Navigator right-click the Business Components project folder and choose **Deploy** and the `ReusableComponents` profile. A Deployment tab appears in the JDeveloper Log window that should display feedback like:

```
---- Deployment started. ---- Apr 28, 2007 7:04:02 PM
Running dependency analysis...
Wrote JAR file to ...\\ReusableComponents\\deploy\\ReusableComponentsCSMT.jar
Running dependency analysis...
Wrote JAR file to ...\\ReusableComponents\\deploy\\ReusableComponentsCSCommon.jar
Elapsed time for deployment: less than one second
---- Deployment finished. ---- Apr 28, 2007 7:04:02 PM
```

33.7.2 How To Import a Package of Reusable Components from a Library

Once you have created a reusable library of business components, you can import one or more packages of components from that library in other projects to reference them. When you import a package of business components from a library, the components in that package are available in the various **Available** lists of the ADF Business Components component wizards and editor, however they do not display in the Application Navigator nor are they editable. The only components that appear in the Application Navigator are the ones in the source path for the current project.

Tip: If you require components that are editable and display in the Application Navigator. Add additional business components from a directory that is not currently part of your project's source path, then open the **Project Content** page of the **Project Properties** dialog and add the parent directory for these other components as one of the directories in the **Java Content** list. In contrast to imported packages of components, additional components added to your project's source path will be fully editable and will appear in the Application Navigator.

To import a package of business components from a library, do the following:

1. Define a library for your JAR file on the **Libraries** tab of the **Project Properties** dialog of the importing project.

You can define the library as a project-level library or a user-level library. Be sure to include both the `*CSMT.jar` and the `*CSCommon.jar` in the class path of the library definition.

2. Include the new library in your importing project's library list.
3. With the importing project selected in the Application Navigator, choose **File | Import** from the JDeveloper main menu.
4. In the **Import** dialog that appears, select **Business Components** from the list.
5. Use the file open dialog to navigate *into* your library's `*CSMT.jar` file — as if it were a directory — and select the XML component definition file from any components in the package whose components you want to import.
6. Acknowledge the alert that confirms the successful importing of the package.
7. Repeat steps 3-6 again for each package of components you want to import.

Assuming that there was an entity object like `Product` in the package(s) of components you imported, you could then create a new view object in the importing project using the imported `Product` component as its entity usage. This is just one example. You can reference any of the imported components as if they were in the source path of your project. The only difference is that you cannot *edit* the imported components. In fact, the reusable component library JAR file might only contain the XML component definition files and the Java `*.class` files for the components without any source code.

33.7.3 How to Remove an Imported Package from a Project

If you mistakenly import a package of components, or wish to remove an imported package of components that you are not using, you can use the Project Properties dialog to do this.

To unimport a package:

1. Double-click the Business Components project in the Application Navigator.
2. In the Project Properties dialog, select **Business Components | Imports**.
3. Select the project from the list of imported projects and click **Delete**.
4. Click **OK** to save the changes to the workspace in JDeveloper.

Caution: Do not remove an imported package if your project still has components that reference it. If you do, JDeveloper will throw exceptions when the project is opened, or your application may have unpredictable behavior. In the Business Components: Imports page of the Project Properties dialog, click the **Show Usages** button to ensure that there are no references to any of the components in the imported package before manually removing the package entry from the * .jpx file.

33.7.4 What Happens When You Import a Package of Reusable Components from a Library

When you import a package of components into a project named *YourImportingProjectName*, JDeveloper adds a reference to that package in the *YourImportingProjectName.jpx* file in the root directory of your importing project's source path. As part of this entry, it includes a design time project named _LocationURL whose value points to the JAR file in which the imported components reside.

33.7.5 What You May Need to Know

When you want to work with a library that you import into your Business Components project, you should be aware of these limitations.

33.7.5.1 Components in Imported Libraries Are Not Editable

If a project imports a package containing business components, the importing project cannot add new components to that same package. The importing project can reference the imported components in new components created in any other package, but cannot add new components to the imported package.

33.7.5.2 Have to Close/Reopen to See Changes from a JAR

If you make changes to your imported components and update the JAR file that contains them, you need to close and reopen any importing projects in order to pickup the changes. This does not require exiting out of JDeveloper. You can select your importing project in the Application Navigator, choose **File | Close** from the main menu, and then re-expand the project's nodes to close and reopen the project. When you do this, JDeveloper will reread the components from the updated version of the imported JAR file.

33.8 Customizing Business Components Error Messages

You can customize any of the built-in ADF Business Components error messages by providing an alternative message string for the error code in a custom message bundle.

33.8.1 How to Customize Base ADF Business Components Error Messages

Assume you want to change the built-in error message:

JBO-27014: Attribute Name is Product is required

If you have requested the Oracle ADF source code from Oracle Worldwide Support, you can look in the `CSMessageBundle.java` file in the `oracle.jbo` package to see that this error message is related to the combination of the following lines in that message bundle file:

```
public class CSMessageBundle extends CheckedListResourceBundle {
    // etc.
    public static final String EXC_VAL_ATTR_MANDATORY           = "27014";
    // etc.
    private static final Object[][][] sMessageStrings = {
        // etc.
        {EXC_VAL_ATTR_MANDATORY, "Attribute {2} in {1} is required"},
        // etc.
    }
}
```

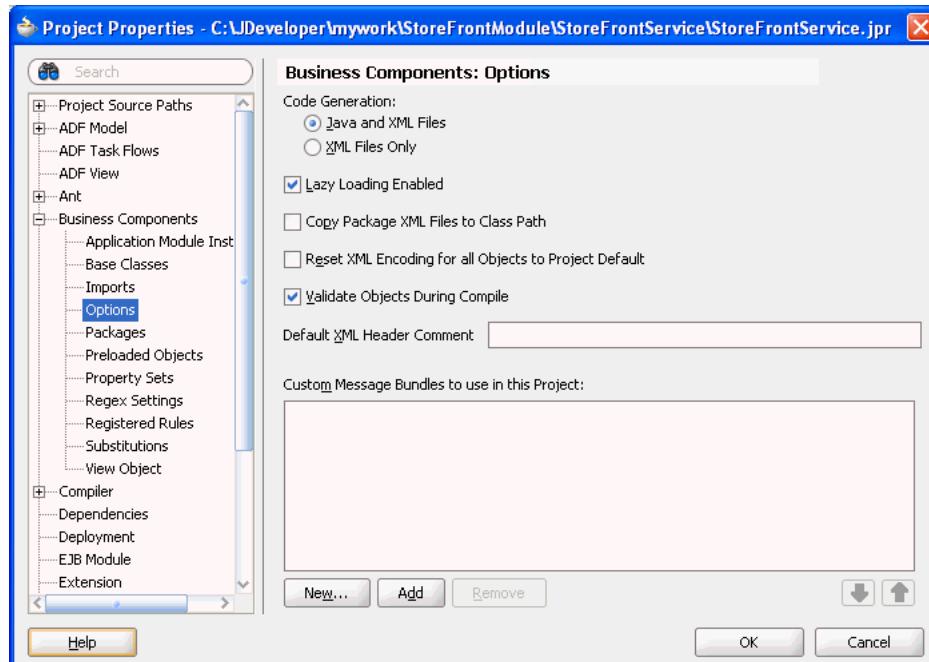
The numbered tokens `{2}` and `{1}` are error message placeholders. In this example the `{1}` is replaced at runtime with the name of the entity object and the `{2}` with the name of the attribute.

To create a custom message bundle file, do the following:

1. Open the **Business Components > Options** page in the **Project Properties** dialog for the project containing your business components.

The **Custom Message Bundles to use in this Project** list displays the bottom of the dialog, as shown in [Figure 33–9](#).

Figure 33–9 Project Properties Displays Message Resource Bundles



2. Click **New...**.

3. Enter a name and package for the custom message bundle in the **Create MessageBundle** class dialog and click **OK**.

Note: If the fully-qualified name of your custom message bundle file does not appear in the **Custom Message Bundles to use in this Project** list, click the **Remove** button, then click the **Add** button to add the new message bundle file created. When the custom message bundle file is correctly registered, its fully-qualified class name should appear in the list.

4. Click **OK** to dismiss the **Project Properties** dialog and open the new custom message bundle class in the source editor.
5. Edit the two-dimensional `String` array in the custom message bundle class to contain any customized messages you'd like to use.

[Example 33–11](#) illustrates a custom message bundle class that overrides the error message string for the JBO-27014 error considered above.

Example 33–11 Custom ADF Business Components Message Bundle

```
package devguide.advanced.customerrs;
import java.util.ListResourceBundle;
public class CustomMessageBundle extends ListResourceBundle {
    private static final Object[][] sMessageStrings
        = new String[][] {
            {"27014", "You must provide a value for {2}"}
        };
    protected Object[][] getContents() {
        return sMessageStrings;
    }
}
```

33.8.2 What Happens When You Customize Base ADF Business Components Error Messages

After adding this message to your custom message bundle file, if you test the application using the Business Component Browser and try to blank out the value of a mandatory attribute, you'll now see your custom error message instead of the default one:

JBO-27014: You must provide a value for Name

You can add as many messages to the message bundle as you want. Any message whose error code key matches one of the built-in error message codes will be used at runtime instead of the default one in the `oracle.jbo.CSMessageBundle` message bundle.

33.8.3 How to Customize Error Messages for Database Constraint Violations

If you enforce constraints in the database, you might want to provide a custom error message in your Fusion web application to display to the end user when one of those constraints is violated. For example, imagine that you added a constraint called `NAME_CANNOT_BEGIN_WITH_X` to the application's `PRODUCTS_BASE` table using the following DDL statement:

```
alter table products_base add (
```

```

constraint name_cannot_begin_with_x
    check (upper(substr(name,1,1)) != 'X')
);

```

To define a custom error message in your application, just add a message to a custom message bundle with the constraint name as the message key. For example, assuming that you use the same `CustomMessageBundle.java` class created in the previous section, [Example 33–12](#) shows what it would look like to define a message with the key `NAME_CANNOT_BEGIN_WITH_X` which matches the name of the database constraint name defined above.

Example 33–12 Customizing Error Message for Database Constraint Violation

```

package devguide.advanced.customerrs;
import java.util.ListResourceBundle;
public class CustomMessageBundle extends ListResourceBundle {
    private static final Object[][] sMessageStrings
        = new String[][] {
            {"27014", "You must provide a value for {2}"},
            {"NAME_CANNOT_BEGIN_WITH_X",
                "The name cannot begin with the letter x!"}
        };
    protected Object[][] getContents() {
        return sMessageStrings;
    }
}

```

33.8.4 How to Implement a Custom Constraint Error Handling Routine

If the default facility for assigning a custom message to a database constraint violation does not meet your needs, you can implement your own custom constraint error handling routine. Doing this requires creating a custom framework extension class for the ADF transaction class, which you then configure your application module to use at runtime.

33.8.4.1 Creating a Custom Database Transaction Framework Extension Class

To write a custom framework extension class for the ADF transaction, create a class like the `CustomDBTransactionImpl` shown in [Example 33–13](#). This example overrides the transaction object's `postChanges()` method to wrap the call to `super.postChanges()` with a `try/catch` block in order to perform custom processing on any `DMLConstraintException` errors that might be thrown. In this simple example, the only custom processing being performed is a call to `ex.setExceptions(null)` to clear out any nested detail exceptions that the `DMLConstraintException` might have. Instead of this, you could perform any other kind of custom exception processing required by your application, including throwing a *custom* exception, provided your custom exception extends `JboException` directly or indirectly.

Example 33–13 Custom Database Transaction Framework Extension Class

```

package devguide.advanced.customerrs;
import oracle.jbo.DMLConstraintException;
import oracle.jbo.JboException;
import oracle.jbo.common.StringManager;
import oracle.jbo.server.DBTransactionImpl2;
import oracle.jbo.server.TransactionEvent;
public class CustomDBTransactionImpl extends DBTransactionImpl2 {
    public void postChanges(TransactionEvent te) {

```

```
try {
    super.postChanges(te);
}
/*
 * Catch the DML constraint exception
 * and perform custom error handling here
 */
catch (DMLConstraintException ex) {
    ex.setExceptions(null);
    throw ex;
}
}
```

33.8.4.2 Configuring an Application Module to Use a Custom Database Transaction Class

In order for your application module to use a custom database transaction class at runtime, you must:

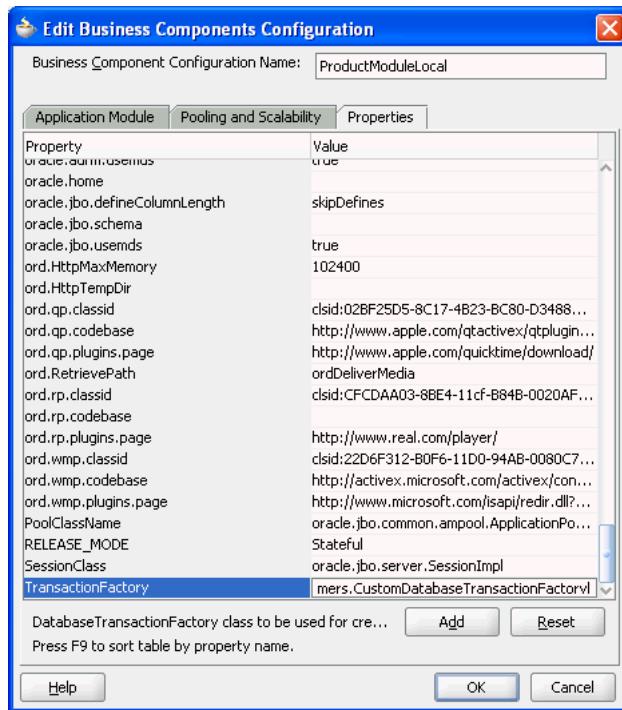
1. Provide a custom implementation of the `DatabaseTransactionFactory` class that overrides the `create()` method to return an instance of the customized transaction class.
2. Configure the value of the `TransactionFactory` property to be the fully-qualified name of this custom transaction factory class.

[Example 33–14](#) shows a custom database transaction factory class that does this. It returns a new instance of the `CustomDBTransactionImpl` class when the framework calls the `create()` method on the database transaction factory.

Example 33–14 Custom Database Transaction Factory Class

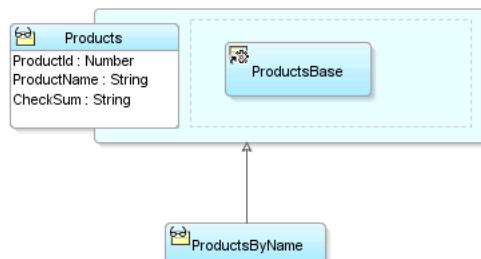
```
package devguide.advanced.customers;
import oracle.jbo.server.DBTransactionImpl2;
import oracle.jbo.server.DatabaseTransactionFactory;
public class CustomDatabaseTransactionFactory
    extends DatabaseTransactionFactory {
    public CustomDatabaseTransactionFactory() {
    }
    /**
     * Return an instance of our custom ToyStoreDBTransactionImpl class
     * instead of the default implementation.
     *
     * @return instance of custom CustomDBTransactionImpl implementation.
     */
    public DBTransactionImpl2 create() {
        return new CustomDBTransactionImpl();
    }
}
```

To complete the job, use the Properties tab of the Create Business Components Configuration editor to assign the value `devguide.advanced.customers.CustomDatabaseTransactionFactory` to the `TransactionFactory` property, as shown in [Figure 33–10](#). You can open the Create Business Components Configuration editor from the Configuration page of the overview editor for the application module by clicking the **Create new configuration objects** button. When you run the application using this configuration, your custom transaction class will be used.

Figure 33–10 ADF Business Components Can Use Custom Database Transaction Class

33.9 Creating Extended Components Using Inheritance

Whenever you create a new business component, if necessary, you can extend an existing one to create a customized version of the original. As shown in [Figure 33–11](#), the `ProductsByName` view object extends the `Products` view object to add a named bind variable named `TheStatus` and to customize the WHERE clause to reference that bind variable.

Figure 33–11 ADF Business Components Can Extend Another Component

While the figure shows a view object example, this component inheritance facility is available for all component types. When one component extends another, the extended component inherits all of the metadata and behavior from the parent it extends. In the extended component, you can add new features or customize existing features of its parent component both through metadata and Java code.

33.9.1 How To Create a Component That Extends Another

To create an extended component, use the component wizard in the **New Gallery** for the type of component you want to create. For example, to create an extended view

object, you use the Create View Object wizard. On the **Name** page of the wizard — in addition to specifying a name and a package for the new component — provide the fully-qualified name of the component that you want to extend in the **Extends** field. To pick the component name from a list, use the **Browse** button next to the **Extends** field. Then, continue to create the extended component in the normal way using the remaining panels of the wizard.

33.9.2 How To Extend a Component After Creation

After defining an extended component, JDeveloper allows you to change the parent component from which an extended component inherits. You can use the component overview editor to accomplish this.

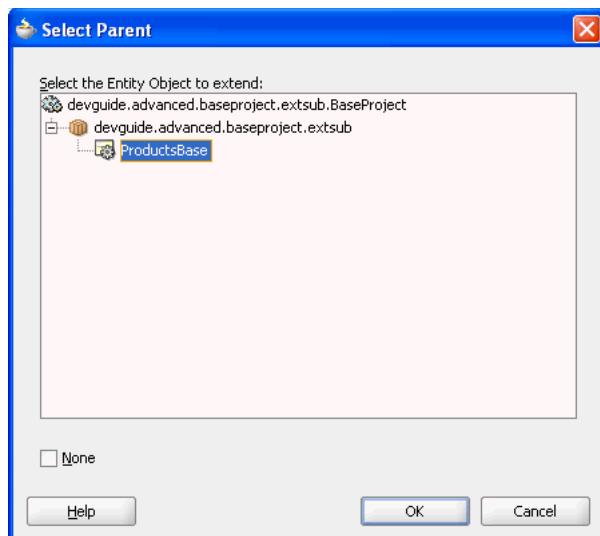
To change the parent component after creation:

1. Open the component in the overview editor and select the General navigation tab.
2. In the displayed editor page, click the **Refactor object extends** button next to the **Extends** field.
3. In the Select Parent dialog, choose the desired component to extend from the package list.

For example, you might extend the `ProductsBase` component, as shown in [Figure 33–12](#).

To change the extended component to not inherit from any parent, select the **None** checkbox in the Select Parent dialog. This has the same effect as if you deleted the component and recreated to accomplish this.

Figure 33–12 Select Parent Dialog



33.9.3 What Happens When You Create a Component That Extends Another

As you've learned, the ADF business components you create are comprised of an XML component definition and an optional Java class. When you create a component that extends another, JDeveloper reflects this component inheritance in both the XML component definition and in any generated Java code for the extended component.

33.9.3.1 Understanding an Extended Component's XML Descriptor

JDeveloper notes the name of the parent component in the new component's XML component definition by adding an `Extends` attribute to the root component element. Any new declarative features you add or any aspects of the parent component's definition you've overridden appear in the extended component's XML component definition. In contrast, metadata that is purely inherited from the parent component is not repeated for the extended component.

[Example 33–15](#) shows what the `ProductsByName.xml` XML component definition for the `ProductsByName` view object looks like. Notice the `Extends` attribute on the `<ViewObject>` element, the `<Variable>` element related to the additional bind variable added in the extended view object, and the overridden value of the `Where` attribute for the WHERE clause that was modified to reference the `theProductName` bind variable.

Example 33–15 Extended Component Reflects Parent in Its XML Descriptor

```
<ViewObject
    xmlns="http://xmlns.oracle.com/bc4j"
    Name="ProductsByName"
    Extends="devguide.advanced.baseproject.extsub.Products"
    Where="UPPER(PRODUCT_NAME) LIKE UPPER(:theProductName) || '%'"
    BindingStyle="OracleName"
    CustomQuery="false"
    RowClass="devguide.advanced.baseproject.extsub.ProductsByNameRowImpl"
    ComponentClass="devguide.advanced.baseproject.extsub.ProductsByNameImpl"
    ...
    <Variable
        Name="theProductName"
        Kind="where"
        Type="java.lang.String"/>
    ...
</ViewObject>
```

33.9.3.2 Understanding Java Code Generation for an Extended Component

If you enable custom Java code for an extended component, JDeveloper automatically generates the Java classes to extend the respective Java classes of its parent component. In this way, the extended component can override any aspect of the parent component's programmatic behavior as necessary. If the parent component is an XML-only component with no custom Java class of its own, the extended component's Java class extends whatever base Java class the parent would use at runtime. This could be the default ADF Business Components framework class in the `oracle.jbo.server` package, or could be your own framework extension class if you have specified that in the `Extends` dialog of the parent component.

In addition, if the extended component is an application module or view object and you enable client interfaces on it, JDeveloper automatically generates the extended component's client interfaces to extend the respective client interfaces of the parent component. If the respective client interface of the parent component does not exist, then the extended component's client interface directly extends the appropriate base ADF Business Components interface in the `oracle.jbo` package.

33.9.4 What You May Need to Know

33.9.4.1 You Can Use Parent Classes and Interfaces to Work with Extended Components

Since an extended component is a customized version of its parent, code you write that works with the *parent* component's Java classes or its client interfaces works without incident for either the parent component *or* any customized version of that parent component.

For example, assume you have a base `Products` view object with custom Java classes and client interfaces like:

- class `ProductsImpl`
- row class `ProductsRowImpl`
- interface `Products`
- row interface `ProductsRow`

If you create a `ProductsByName` view object that extends `Products`, then you can use the base component's classes and interface to work both with `Products` and `ProductsByName`.

[Example 33–16](#) illustrates a test client program that works with the `Products`, `ProductsRow`, `ProductsByName`, and `ProductsByNameRow` client interfaces. A few interesting things to note about the example are the following:

1. You can use parent `Products` interface for working with the `ProductsByName` view object that extends it.
2. Alternatively, you can cast an instance of the `ProductsByName` view object to its own more specific `ProductsByName` client interface.
3. You can test if row `ProductsRow` is actually an instance of the more specific `ProductsByNameRow` before casting it and invoking a method specific to the `ProductsByNameRow` interface.

Example 33–16 Working with Parent and Extended Components

```
package devguide.advanced.baseproject.extsub;
/* imports omitted */
public class TestClient {
    public static void main(String[] args) {
        String      amDef = "devguide.advanced.baseproject.extsub.ProductModule";
        String      config = "ProductModuleLocal";
        ApplicationModule am =
            Configuration.createRootApplicationModule(amDef, config);
        Products products = (Products)am.findViewObject("Products");
        products.executeQuery();
        ProductsRow product = (ProductsRow)products.first();
        printAllAttributes(products,product);
        testSomethingOnProductsRow(product);
        // 1. You can use parent Products interface for ProductsByName
        products = (Products)am.findViewObject("ProductsById");
        // 2. Or cast it to its more specific ProductsByName interface
        ProductsByName productsById = (ProductsByName)products;
        productsById.setProductName("Ice");
        productsById.executeQuery();
        product = (ProductsRow)productsById.first();
        printAllAttributes(productsById,product);
        testSomethingOnProductsRow(product);
        am.getTransaction().rollback();
        Configuration.releaseRootApplicationModule(am,true);
    }
}
```

```

}
private static void testSomethingOnProductsRow(ProductsRow product) {
    try {
        // 3. Test if row is a ProductsByNameRow before casting
        if (product instanceof ProductsByNameRow) {
            ProductsByNameRow productByName = (ProductsByNameRow)product;
            productByName.someExtraFeature("Test");
        }
        product.setName("Q");
        System.out.println("Setting the Name attribute to 'Q' succeeded.");
    }
    catch (ValidationException v) {
        System.out.println(v.getLocalizedMessage());
    }
}
private static void printAllAttributes(ViewObject vo, Row r) {
    String viewObjName = vo.getName();
    System.out.println("Printing attribute for a row in VO '" +
        viewObjName+"\"");
    StructureDef def = r.getStructureDef();
    StringBuilder sb = new StringBuilder();
    int numAttrs = def.getAttributeCount();
    AttributeDef[] attrDefs = def.getAttributeDefs();
    for (int z = 0; z < numAttrs; z++) {
        Object value = r.getAttribute(z);
        sb.append(z > 0 ? " " : "")
            .append(attrDefs[z].getName())
            .append(">")
            .append(value == null ? "<null>" : value)
            .append(z < numAttrs - 1 ? "\n" : "");
    }
    System.out.println(sb.toString());
}
}

```

Running the test client above produces the following results:

```

Printing attribute for a row in VO 'Products'
ProdId=1
    ProductName=Plasma HD Television
    Checksum=I am the Product Class
Setting the Name attribute to 'Q' succeeded.
Printing attribute for a row in VO 'ProductsByName'
ProdId=15
    ProductName=Ipod Speakers
    Checksum=I am the Product Class
    SomeExtraAttr=<null>
## Called someExtraFeature of ProductsByNameRowImpl
Setting the Name attribute to 'Q' succeeded.

```

Note: In this example, Products is an entity-based view object based on the Product entity object. The Product entity object includes a transient Checksum attribute that returns the string "I am the Product class". You'll learn more about why this was included in the example in [Section 33.10, "Substituting Extended Components In a Delivered Application"](#).

33.9.4.2 Class Extends is Disabled for Extended Components

When you create an extended component, the **Class Extends** button on the **Java** page of the extended component's wizard is disabled. Additionally, in the application module editor's Java page, when you click **Edit java options**, the **Class Extends** button in the Java dialog appears disabled. This is due to the fact that JDeveloper automatically extends the appropriate class of its parent component, so it does not make sense to allow you to select a different class.

33.9.4.3 Interesting Aspects You Can Extend for Key Component Types

Entity Objects

When you create an extended entity object, you can introduce new attributes, new associations, new validators, and new custom code. You can override certain declarative aspects of existing attributes as well as overriding any method from the parent component's class.

View Objects

When you create an extended view object, you can introduce new attributes, new view links, new bind variables, and new custom code. You can override certain declarative aspects of existing attributes as well as overriding any method from the parent component's class.

Application Modules

When you create an extended application module, you can introduce new view object instances or new nested application module instance and new custom code. You can also override any method from the parent component's class.

33.9.4.4 Extended Components Have Attribute Indices Relative to Parent

If you add new attributes in an extended entity object or view object, the attribute index numbers are computed relative to the parent component. For example, consider the `Products` view object mentioned above. If you enable a custom view row class, it might have attribute index constants defined in the `ProductsRowImpl.java` class like this:

```
public class ProductsRowImpl extends ViewRowImpl
    implements ProductsRow {
    public static final int PRODID = 0;
    public static final int NAME = 1;
    public static final int CHECKSUM = 2;
    //etc.
}
```

When you create an extended view object like `ProductsByName`, if that view object adds an addition attribute like `SomeExtraAttr` and has a custom view row class enabled, then its attribute constants will be computed relative to the maximum value of the attribute constants in the parent component:

```
public class ProductsByNameRowImpl extends ProductsRowImpl
    implements ProductsByNameRow {
    public static final int MAXATTRCONST =
        ViewDefImpl.getMaxAttrConst("devguide.advanced.baseproject.extsub.Products");
    public static final int SOMEEXTRAATTR = MAXATTRCONST;
```

Additional attributes would have index values of `MAXATTRCONST+1`, `MAXATTRCONST+2`, etc.

33.10 Substituting Extended Components In a Delivered Application

If you deliver packaged applications that can require on-site customization for each potential client of your solution, ADF Business Components offers a useful feature to simplify that task.

33.10.1 Extending and Substituting Components Is Superior to Modifying Code

All too often, on-site application customization is performed by making direct changes to the source code of the delivered application. This approach demonstrates its weaknesses whenever you deliver patches or new feature releases of your original application to your clients. Any customizations they had been applied to the base application's source code need to be painstakingly re-applied to the patched or updated version of the base application. Not only does this render the application customization a costly, ongoing maintenance expense, it can introduce subtle bugs due to human errors that occur when reapplying previous customizations to new releases.

ADF Business Components offers a superior, component-based approach to support application customization that doesn't require changing — or even having access to — the base application's source code. To customize your delivered application, your customers can:

1. Import one or more packages of components from the base application into a new project.
2. Create new components to effect the application customization, extending appropriate parent components from the base application as necessary.
3. Define a list of global component substitutions, naming their customized components to substitute for your base application's appropriate parent components.

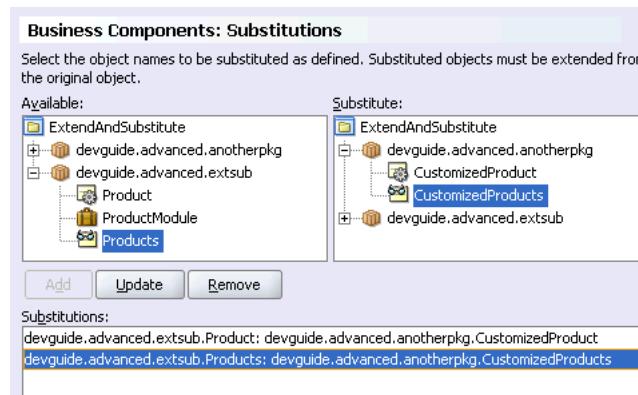
When the customer runs your delivered application with a global component substitution list defined, their customized application components are used by your delivered application without changing any of its code. When you deliver a patched or updated version of the original application, their component customizations apply to the updated version the next time they restart the application without needing to re-apply any customizations.

33.10.2 How To Substitute an Extended Component

To define global component substitutions, use the **Business Components > Substitutions** page of the **Project Properties** dialog in the project where you've created extended components based on the imported components from the base application. As shown in [Figure 33-13](#), to define each component substitution:

1. Select the base application's component in the **Available** list.
2. Select the customized, extended component to substitute in the **Substitute** list.
3. Click **Add**.

Note: You can only substitute a component in the base application with an extended component that inherits directly or indirectly from the base one.

Figure 33–13 Defining Business Components Substitutions

33.10.3 What Happens When You Substitute

When you define a list of global component substitutions in a project named *YourExtendsAndSubstitutesProject*, the substitution list is saved in the *YourExtendsAndSubstitutesProject.jpx* in the root directory of the source path.

The file will contain <Substitute> elements as shown in [Example 33–17](#), one for each component to be substituted.

Example 33–17 Component Substitution List Saved in the Project's JPX File

```
<JboProject
    Name="ExtendAndSubstitute"
    SeparateXMLFiles="true"
    PackageName="" >
    <Containee
        Name="anotherpkg"
        FullName="devguide.advanced.anotherpkg.anotherpkg"
        ObjectType="JboPackage" >
    </Containee>
    <Containee
        Name="extsub"
        FullName="devguide.advanced.extsub"
        ObjectType="JboPackage" >
        <DesignTime>
            <Attr Name="_LocationURL"
                Value="../../BaseProject/deploy/BaseProjectCSMT.jar" />
        </DesignTime>
    </Containee>
    <Substitutes>
        <Substitute OldName="devguide.advanced.extsub.Product"
                    NewName="devguide.advanced.anotherpkg.CustomizedProduct" />
        <Substitute OldName="devguide.advanced.extsub.Products"
                    NewName="devguide.advanced.anotherpkg.CustomizedProducts" />
    </Substitutes>
</JboProject>
```

33.10.4 Enabling the Substituted Components in the Base Application

To have the original application use the set of substituted components, define the Java system property `Factory-Substitution-List` and set its value to the name of the

project whose *.jpx file contains the substitution list. The value should be just the project name without any *.jpr or *.jpx extension.

Consider a simple example that customizes the Product entity object and the Products view object described in [Section 33.9.4.1, "You Can Use Parent Classes and Interfaces to Work with Extended Components"](#). To perform the customization, assume you create new project named ExtendsAndSubstitutes that:

- Defines a library for the JAR file containing the base components
- Imports the package containing Product and Products
- Creates new extended components in a distinct package name called CustomizedProduct and CustomizedProducts
- Defines a component substitution list to use the extended components.

When creating the extended components, assume that you:

- Added an extra view attribute named ExtraViewAttribute to the CustomizedProducts view object.
- Added a new validation rule to the CustomizedProduct entity object to enforce that the product name cannot be the letter "Q".
- Overrode the getChecksum() method in the CustomizedProduct.java class to return "**I am the CustomizedProduct Class**".

If you define the Factory-Substitution-List Java system property set to the value ExtendsAndSubstitutes, then when you run the exact same test client class shown above in [Example 33-16](#) the output of the sample will change to reflect the use of the substituted components:

```
Printing attribute for a row in VO 'Products'
ProdId=1
    ProductName=Plasma HD Television
    Checksum=I am the CustomizedProduct Class
    ExtraViewAttribute=Extra Attr Value
The name cannot be Q!
Printing attribute for a row in VO 'ProductsByName'
ProdId=15
    ProductName=IPod Speakers
    Checksum=I am the CustomizedProduct Class
    SomeExtraAttr=SomeExtraAttrValue
## Called someExtraFeature of ProductsByNameRowImpl
The name cannot be Q!
```

Compared to the output from [Example 33-16](#), notice that in the presence of the factory substitution list, the Products view object in the original test program now has the additional ExtraViewAttribute, now reports a Checksum attribute value of "I am the CustomizedProduct Class", and now disallows the assignment of the product name to have the value "Q". These component behavior changes were performed without needing to modify the original Java or XML source code of the delivered components.

34

Advanced Entity Object Techniques

This chapter describes advanced techniques for use in the entity objects in your business domain layer.

This chapter includes the following sections:

- [Section 34.1, "Creating Custom, Validated Data Types Using Domains"](#)
- [Section 34.2, "Updating a Deleted Flag Instead of Deleting Rows"](#)
- [Section 34.3, "Using Update Batching"](#)
- [Section 34.4, "Advanced Entity Association Techniques"](#)
- [Section 34.5, "Basing an Entity Object on a PL/SQL Package API"](#)
- [Section 34.6, "Basing an Entity Object on a Join View or Remote DBLink"](#)
- [Section 34.7, "Using Inheritance in Your Business Domain Layer"](#)
- [Section 34.8, "Controlling Entity Posting Order to Avoid Constraint Violations"](#)
- [Section 34.9, "Implementing Automatic Attribute Recalculation"](#)
- [Section 34.10, "Implementing Custom Validation Rules"](#)

Note: To experiment with a working version of the examples in this chapter, download the AdvancedEntityExamples workspace from the *Example Downloads* page at http://otn.oracle.com/documentation/jdev/b25947_01/.

34.1 Creating Custom, Validated Data Types Using Domains

When you find yourself repeating the same sanity-checking validations on the values of similar attributes across multiple entity objects, you can save yourself time and effort by creating your own data types that encapsulate this validation. For example, imagine that across your business domain layer there are numerous entity object attributes that store strings that represent email addresses. One technique you could use to ensure that end-users always enter a valid email address everywhere one appears in your business domain layer is to:

- Use a basic `String` data type for each of these attributes
- Add an attribute-level method validator with Java code that ensures that the `String` value has the format of a valid email address for each attribute

However, these approaches can become tedious quickly in a large application. Fortunately, ADF Business Components offers an alternative that allows you to create your own `EmailAddress` data type that represents an email address. After

centralizing all of the sanity-checking regarding email address values into this new custom data type, you can use the `EmailAddress` as the type of every attribute in your application that represents an email address. By doing this, you make the intention of the attribute values more clear to other developers and simplify application maintenance by putting the validation in a single place. ADF Business Components calls these developer-created data types domains.

Note: The examples in this section refer to the `SimpleDomains` project in the `AdvancedEntityExamples` workspace. See the note at the beginning of this chapter for download instructions. Run the `CreateObjectType.sql` script in the **Resources** folder against the FOD connection to set up the additional database objects required for the project.

34.1.1 What Are Domains?

Domains are Java classes that extend the basic data types like `String`, `Number`, and `Date` to add constructor-time validation to insure the candidate value passes relevant sanity checks. They offer you a way to define custom data types with cross-cutting behavior such as basic data type validation, formatting, and custom metadata properties in a way that are inherited by any entity objects or view objects that use the domain as the Java type of any of their attributes.

34.1.2 How To Create a Domain

To create a domain, use the Create Domain wizard. This is available in the **New Gallery** in the **ADF Business Components** category.

To create a domain:

1. Launch the Create Domain wizard.
2. On the **Name** page, specify a name for the domain and a package in which it will reside. To create a domain based on a simple Java type, leave **Domain for an Oracle Object Type** unchecked.
3. Click **Next**.
4. On the **Settings** page, indicate the base type for the domain and the database column type to which it will map. For example, if you were creating a domain called `ShortEmailAddress` to hold eight-character short email addresses, you would set the base type to `String` and the **Database Column Type** to `VARCHAR2 (8)`. You can set other common attribute settings on this panel as well.
5. Click **Finish** to create your domain.

34.1.3 What Happens When You Create a Domain

When you create a domain, JDeveloper creates its XML component definition in the subdirectory of your project's source path that corresponds to the package name you chose. For example, if you created the `ShortEmailAddress` domain in the `devguide.advanced.domains` package, JDeveloper would create the `ShortEmailAddress.xml` file in the `./devguide/advanced/domains` subdirectory. A domain always has a corresponding Java class, which JDeveloper creates in the common subpackage of the package where the domain resides. This means it would create the `ShortEmailAddress.java` class in the `devguide.advanced.domains.common` package. The domain's Java class is

automatically generated with the appropriate code to behave in a way that is identical to one of the built-in data types.

34.1.4 What You May Need to Know About Domains

The sections that follow describe some of the things you may need to know about when using domains.

34.1.4.1 Using Domains for Entity and View Object Attributes

Once you've created a domain in a project, it automatically appears among the list of available data types in the **Attribute Type** dropdown list in the entity object and view object wizards and editors. To use the domain as the type of a given attribute, just pick it from the list.

Note: The entity-mapped attributes in an entity-based view object inherit their data type from their corresponding underlying entity object attribute, so if the entity attribute uses a domain type, so will the matching view object attribute. For transient or SQL-derived view object attributes, you can directly set the type to use a domain since it is not inherited from any underlying entity.

34.1.4.2 Validate Method Should Throw DataCreationException If Sanity Checks Fail

Typically, the only coding task you need to do for a domain is to write custom code inside the generated validate() method. Your implementation of the validate() method should perform your sanity checks on the candidate value being constructed, and throw a DataCreationException in the oracle.jbo package if the validation fails.

In order to throw an exception message that is translatable, you can create a message bundle class similar to the one shown in [Example 34-1](#). Create it in the same common package as your domain classes themselves. The message bundle returns an array of {MessageKeyString, TranslatableMessageString} pairs.

Example 34-1 Custom Message Bundle Class For Domain Exception Messages

```
package devguide.advanced.simpledomains.common;

import java.util.ListResourceBundle;

public class ErrorMessages extends ListResourceBundle {
    public static final String INVALID_SHORTEMAIL = "30002";
    public static final String INVALID_EVENNUMBER = "30003";
    private static final Object[][] sMessageStrings = new String[][] {
        { INVALID_SHORTEMAIL,
            "A valid short email address has no @-sign or dot." },
        { INVALID_EVENNUMBER,
            "Number must be even." }
    };

    /**
     * Return String Identifiers and corresponding Messages
     * in a two-dimensional array.
     */
    protected Object[][] getContents() {
        return sMessageStrings;
    }
}
```

```

    }
}

```

34.1.4.3 String Domains Aggregate a String Value

Since `String` is a base JDK type, a domain based on a `String` aggregates a private `mData` `String` member field to hold the value that the domain represents. Then, the class implements the `DomainInterface` expected by the ADF runtime, as well as the `Serializable` interface, so the domain can be used in method arguments or returns types of ADF components custom client interfaces.

[Example 34–2](#) shows the `validate()` method for a simple `ShortEmailAddress` domain class. It tests to make sure that the `mData` value does not contain an at-sign or a dot, and if it does, then the method throws `DataCreationException` referencing an appropriate message bundle and message key for the translatable error message.

Example 34–2 Simple ShortEmailAddress String-Based Domain Type with Custom Validation

```

public class ShortEmailAddress implements DomainInterface, Serializable {
    private String mData;
    // etc.
    /**Implements domain validation logic and throws a JboException on error. */
    protected void validate() {
        int atpos = mData.indexOf('@');
        int dotpos = mData.lastIndexOf('.');
        if (atpos > -1 || dotpos > -1) {
            throw new DataCreationException(ErrorMessages.class,
                ErrorMessages.INVALID_SHORTEMAIL,null,null);
        }
    }
    // etc.
}

```

34.1.4.4 Other Domains Extend Existing Domain Type

Other simple domains based on a built-in type in the `oracle.jbo.domain` package extend the base type as shown in [Example 34–3](#). It illustrates the `validate()` method for a simple Number-based domain called `EvenNumber` that represents even numbers.

Example 34–3 Simple EvenNumber Number-Based Domain Type with Custom Validation

```

public class EvenNumber extends Number {
    // etc.
    /**
     * Validates that value is an even number, or else
     * throws a DataCreationException with a custom
     * error message.
     */
    protected void validate() {
        if (getValue() % 2 == 1) {
            throw new DataCreationException(ErrorMessages.class,
                ErrorMessages.INVALID_EVENNUMBER,null,null);
        }
    }
    // etc.
}

```

34.1.4.5 Simple Domains are Immutable Java Classes

When you create a simple domain based on one of the basic data types, it is an *immutable* class. This just means that once you've constructed a new instance of it like this:

```
ShortEmailAddress email = new ShortEmailAddress("smuench");
```

You cannot change its value. If you want to reference a different short email address, you just construct another one:

```
ShortEmailAddress email = new ShortEmailAddress("bribet");
```

This is not a new concept since it's the same way that `String`, `Number`, and `Date` classes behave, among others.

34.1.4.6 Creating Domains for Oracle Object Types When Useful

The Oracle database supports the ability to create user-defined types in the database. For example, you could create a type called `POINT_TYPE` using the following DDL statement:

```
create type point_type as object (
    x_coord number,
    y_coord number
);
```

If you use user-defined types like `POINT_TYPE`, you can create domains based on them, or you can reverse-engineer tables containing columns of object type to have JDeveloper create the domain for you.

Manually Creating Oracle Object Type Domains

To create a domain yourself, perform the following steps in the Create Domain wizard:

1. Launch the Create Domain wizard.
2. On the **Name** page, check the **Domain for an Oracle Object Type** checkbox, then select the object type for which you want to create a domain from the **Available Types** list.
3. Click **Next**.
4. On the **Settings** page, use the **Attribute** dropdown list to switch between the multiple domain properties to adjust the settings as appropriate.
5. Click **Finish** to create the domain.

Reverse-Engineering Oracle Object Type Domains

In addition to manually creating object type domains, when you use the Business Components from Tables wizard and select a table containing columns of an Oracle object type, JDeveloper creates domains for those object types as part of the reverse-engineering process. For example, imagine you created a table like this with a column of type `POINT_TYPE`:

```
create table interesting_points(
    id number primary key,
    coordinates point_type,
    description varchar2(20)
);
```

If you create an entity object for the `INTERESTING_POINTS` table in the Business Components from Tables wizard, then you will get both an `InterestingPoints`

entity object and a PointType domain. The latter will have been automatically created, based on the `POINT_TYPE` object type, since it was required as the data type of the `Coordinates` attribute of the `InterestingPoints` entity object.

Unlike simple domains, object type domains are mutable. JDeveloper generates getter and setter methods into the domain class for each of the elements in the object type's structure. After changing any domain properties, when you set that domain as the value of a view object or entity object attribute, it is treated as a single unit. ADF does not track which domain properties have changed, only that a domain-valued attribute value has changed.

Note: Domains based on Oracle object types are useful for working programmatically with data whose underlying type is an oracle object type. They also can simplify passing and receiving structure information to stored procedures. However, support for working with object type domains in the ADF binding layer is complete, so it's not straightforward to use object domain-valued attributes in declaratively-databound user interfaces.

34.1.4.7 Quickly Navigating to the Domain Class

After selecting a domain in the Application Navigator, you can quickly navigate to its implementation class by:

- Choosing **Go to Domain Class** on the right-mouse context menu, or
- Double-clicking on the domain class in the Structure Window

34.1.4.8 Domains Get Packaged in the Common JAR

When you create a business components archive, as described in [Section 33.7, "Working with Libraries of Reusable Business Components"](#), the domain classes and message bundle files in the `*.common` subdirectories of your project's source path get packaged into the `*CSCommon.jar`. They are classes that are common to both the middle-tier application server and to an eventual remote-client you might need to support.

34.1.4.9 Entity and View Object Attributes Inherit Custom Domain Properties

You can define custom metadata properties on a domain. Any entity object or view object attribute based on that domain inherits those custom properties as if they had been defined on the attribute itself. If the entity object or view object attribute defines the same custom property, its setting takes precedence over the value inherited from the domain.

34.1.4.10 Domain Settings Cannot Be Less Restrictive at Entity or View Level

JDeveloper enforces the declarative settings you impose at the domain definition level; they cannot be made *less* restrictive for the entity object or view object for an attribute based on the domain type. For example, if you define a domain to have its **Updatable** property set to **While New**, then when you use your domain as the Java type of an entity object attribute, you can set **Updatable** to be **Never** (more restrictive) but you cannot set it to be **Always**. Similarly, if you define a domain to be **Persistent**, you cannot make it transient later. When sensible for your application, set declarative properties for a domain to be as lenient as possible so you can later make them more restrictive as needed.

34.2 Updating a Deleted Flag Instead of Deleting Rows

For auditing purposes, once a row is added to a table, sometimes your requirements may demand that rows are never physically deleted from the table. Instead, when the end-user deletes the row in the user interface, the value of a `DELETED` column should be updated from "N" to "Y" to mark it as deleted. This section explains the two method overrides required to alter an entity object's default behavior to achieve this effect. The following sections assume you want to change the `Products` entity from the Fusion Order Demo application to behave in this way. They presume that you've altered the `PRODUCTS` table to have an additional `DELETED` column, and synchronized the `Products` entity with the database to add the corresponding `Deleted` attribute.

34.2.1 How to Update a Deleted Flag When a Row is Removed

To update a deleted flag when a row is removed, enable a custom Java class for your entity object and override the `remove()` method to set the deleted flag before calling the `super.remove()` method. [Example 34-4](#) shows what this would look like in the `ProductsImpl` class of the Fusion Order Demo application's `Products` entity object. It is important to set the attribute *before* calling `super.remove()` since an attempt to set the attribute of a deleted row will encounter the `DeadEntityAccessException`.

Example 34-4 Updating a Deleted Flag When a Products Entity Row is Removed

```
// In ProductsImpl.java
public void remove() {
    setDeleted("Y");
    super.remove();
}
```

The row will still be removed from the row set, but it will have the value of its `Deleted` flag modified to "Y" in the entity cache. The second part of implementing this behavior involves forcing the entity to perform an `UPDATE` instead of an `INSERT` when it is asked to perform its DML operation. You need to implement both parts for a complete solution.

34.2.2 Forcing an Update DML Operation Instead of a Delete

To force an entity object to be updated instead of deleted, override the `doDML()` method and write code that conditionally changes the `operation` flag. When the `operation` flag equals `DML_DELETE`, your code will change it to `DML_UPDATE` instead. [Example 34-5](#) shows what this would look like in the `ProductsImpl` class of the Fusion Order Demo application's `Products` entity object.

Example 34-5 Forcing an Update DML Operation Instead of a Delete

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

With this overridden `doDML()` method in place to complement the overridden `remove()` method described in the previous section, any attempt to remove a `Product` entity through any view object with a `Products` entity usage will update the `DELETED` column instead of physically deleting the row. Of course, in order to prevent "deleted" products from appearing in your view object query results, you will

need to appropriately modify their WHERE clauses to include only products WHERE DELETED = 'N'.

34.3 Using Update Batching

You can use update batching to reduce the number of DML statements issued with multiple entity modifications.

By default, the ADF Business Components framework performs a single DML statement (INSERT, UPDATE, DELETE) for each modified entity of a given entity definition type. For example, say you have an Employee entity object type for which multiple instances are modified during typical use of the application. If two instances were created, three existing instances modified, and four existing instances deleted, then at transaction commit time the framework issues nine DML statements (2 INSERTS, 3 UPDATES, and 4 DELETES) to save these changes.

If you will frequently be updating more than one entity of a given type in a transaction, consider using the update batching feature for that entity definition type. In the example, update batching (with a threshold of 1) causes the framework to issue just three DML statements: one bulk INSERT statement processing two inserts, one bulk UPDATE statement processing three updates, and one bulk DELETE statement processing four deletes.

Note: If the entity object has any attributes that are set to **Refresh After Insert** or **Refresh After Update**, then the batch update feature is disabled.

To enable update batching for an entity

1. Open the appropriate entity object in the overview editor.
2. In the Tuning section of the General page, select the **Use Update Batching** checkbox, and specify the appropriate threshold.

This establishes a batch processing threshold beyond which ADF will process the modifications in a bulk DML operation.

34.4 Advanced Entity Association Techniques

This section describes several advanced techniques for working with associations between entity objects.

34.4.1 Modifying Association SQL Clause to Implement Complex Associations

When you need to represent a more complex relationship between entities than one based only on the equality of matching attributes, you can modify the association's SQL clause to include more complex criteria. For example, sometimes the relationship between two entities depends on effective dates. A Product may be related to a Supplier, however if the name of the supplier changes over time, each row in the SUPPLIERS table might include additional EFFECTIVE_FROM and EFFECTIVE_UNTIL columns that track the range of dates in which that product row is (or was) in use. The relationship between a Product and the Supplier with which it is associated might then be described by a combination of the matching SupplierId attributes and a condition that the product's RequestDate lie between the supplier's EffectiveFrom and EffectiveUntil dates.

You can setup this more complex relationship in the overview editor for the association. First add any additional necessary attribute pairs on the **Relationship** page, which in this example would include one (`EffectiveFrom`, `RequestDate`) pair and one (`EffectiveUntil`, `RequestDate`) pair. Then on the **Query** page you can edit the **Where** field to change the WHERE clause to be:

```
(:Bind_SupplierId = Product.SUPPLIER_ID) AND
(Product.REQUEST_DATE BETWEEN :Bind_EffectiveFrom
                           AND :Bind_EffectiveUntil)
```

For more information about creating associations, see [Section 4.3, "Creating and Configuring Associations"](#).

34.4.2 Exposing View Link Accessor Attributes at the Entity Level

When you create a view link between two entity-based view objects, on the **View Link Properties** page, you have the option to expose view link accessor attributes both at the view object level as well as at the entity object level. By default, a view link accessor is only exposed at the view object level of the destination view object. By checking the appropriate **In Entity Object: SourceEntityName** or **In Entity Object: DestinationEntityName** checkbox, you can opt to have JDeveloper include a view link attribute in either or both of the source or destination entity objects. This can provide a handy way for an entity object to access a related set of related view rows, especially when the query to produce the rows only depends on attributes of the current row.

34.4.3 Optimizing Entity Accessor Access By Retaining the Row Set

Each time you retrieve an entity association accessor row set, by default the entity object creates a new `RowSet` object to allow you to work with the rows. This does *not* imply re-executing the query to produce the results each time, only creating a new instance of a `RowSet` object with its default iterator reset to the "slot" before the first row. To force the row set to refresh its rows from the database, you can call its `executeQuery()` method.

Since there is a small amount of overhead associated with creating the row set, if your code makes numerous calls to the same association accessor attributes, you can consider enabling the association accessor row set retention for the source entity object in the association. You can enable retention of the association accessor row set using the overview editor for the entity object that is the source for the association accessor. Select **Retain Association Accessor Row Set** in the Tuning section of the General page of the overview editor for the entity object.

Alternatively, you can enable a custom Java *entity collection* class for your entity object. As with other custom entity Java classes you've seen, you do this on the Select Java Options dialog that you open from the Java page of the overview editor for the entity object. In the dialog, select **Generate Entity Collection Class**. Then, in the `YourEntityCollImpl` class that JDeveloper creates for you, override the `init()` method, and add a line after `super.init()` that calls the `setAssociationAccessorRetained()` method passing `true` as the parameter. It affects all association accessor attributes for that entity object.

When this feature is enabled for an entity object, since the association accessor row set is not recreated each time, the current row of its default row set iterator is also retained as a side-effect. This means that your code will need to explicitly call the `reset()` method on the row set you retrieve from the association accessor to reset the current row in its default row set iterator back to the "slot" before the first row.

Note, however, that with accessor retention enabled, your failure to call `reset()` each time before you iterate through the rows in the accessor row set can result in a subtle, hard-to-detect error in your application. For example, if you iterate over the rows in an association accessor row set like this, for example, to calculate some aggregate total:

Example 34–6 Iterating over a row set incorrectly

```
// In SuppliersImpl.java
RowSet rs = (RowSet)getProducts();
while (rs.hasNext()) {
    ProductImpl r = (ProductImpl)rs.next();
    // Do something important with attributes in each row
}
```

The first time you work with the accessor row set, the code will work. However, since the row set (and its default row set iterator) are retained, the second and subsequent times you access the row set the current row will already be at the end of the row set and the while loop will be skipped since `rs.hasNext()` will be `false`. Instead, with this feature enabled, write your accessor iteration code like this:

Example 34–7 Iterating over a row set and resetting to the first row

```
// In SuppliersImpl.java
RowSet rs = (RowSet)getProducts();
rs.reset(); // Reset default row set iterator to slot before first row!
while (rs.hasNext()) {
    ProductImpl r = (ProductImpl)rs.next();
    // Do something important with attributes in each row
}
```

34.5 Basing an Entity Object on a PL/SQL Package API

If you have a PL/SQL package that encapsulates insert, update, and delete access to an underlying table, you can override the default DML processing event for the entity object that represents that table to invoke the procedures in your PL/SQL API instead. Often, such PL/SQL packages are used in combination with a companion database view. Client programs read data from the underlying table using the database view, and "write" data back to the table using the procedures in the PL/SQL package. This section considers the code necessary to create a `Product` entity object based on such a combination of a view and a package.

Given the `PRODUCTS` table in the Fusion Order Demo schema, consider a database view named `PRODUCTS_V`, created using the following DDL statement:

```
create or replace view products_v
as select product_id,name,image,description from products;
```

In addition, consider the simple `PRODUCTS_API` package shown in [Example 34–8](#) that encapsulates insert, update, and delete access to the underlying `PRODUCTS` table.

Example 34–8 Simple PL/SQL Package API for the PRODUCTS Table

```
create or replace package products_api is
    procedure insert_product(p_product_id number,
                            p_name varchar2,
                            p_image varchar2,
                            p_description varchar2);
    procedure update_product(p_product_id number,
```

```

        p_name varchar2,
        p_image varchar2,
        p_description varchar2);
procedure delete_product(p_product_id number);
end products_api;

```

The following sections explain how to create an entity object based on the above combination of view and package.

Note: The examples in this section refer to the EntityWrappingPLSQLPackage project in the AdvancedEntityExamples workspace. See the note at the beginning of this chapter for download instructions. Run the CreateXXX.sql scripts in the `src` folder against the FOD connection to set up the additional database objects required for the project.

34.5.1 How to Create an Entity Object Based on a View

To create an entity object based on a view, use the Create Entity Object wizard and perform the following steps:

- On the **Name** page, give the entity a name like `Product` and check the **Views** checkbox at the bottom of the **Database Objects** section.
This enables the display of the available database views in the current schema in the **Schema Object** list.
- Select the desired database view in the **Schema Object** list.
- On the **Attribute Settings** page, use the **Select Attribute** dropdown list to choose the attribute that will act as the primary key, then enable the **Primary Key** setting for that property.

Note: When defining the entity based on a view, JDeveloper cannot automatically determine the primary key attribute since database views do not have related constraints in the database data dictionary.

- Then click **Finish**.

34.5.2 What Happens When You Create an Entity Object Based on a View

By default, an entity object based on a view performs all of the following directly against the underlying database view:

- `SELECT` statement (for `findByPrimaryKey()`)
- `SELECT FOR UPDATE` statement (for `lock()`), and
- `INSERT, UPDATE, DELETE` statements (for `doDML()`)

The following sections first illustrate how to override the `doDML()` operations, then explain how to extend that when necessary to override the `lock()` and `findByPrimaryKey()` handling in a second step.

34.5.3 Centralizing Details for PL/SQL-Based Entities into a Base Class

If you plan to have more than one entity object based on a PL/SQL API, it's a smart idea to abstract the generic details into a base framework extension class. In doing this,

you'll be using several of the concepts you learned in [Chapter 33, "Advanced Business Components Techniques"](#). Start by creating a `PLSQUEntityImpl` class that extends the base `EntityImpl` class that each one of your PL/SQL-based entities can use as their base class. As shown in [Example 34–9](#), you'll override the `doDML()` method of the base class to invoke a different helper method based on the operation.

Example 34–9 Overriding `doDML()` to Call Different Procedures Based on the Operation

```
// In PLSQUEntityImpl.java
protected void doDML(int operation, TransactionEvent e) {
    // super.doDML(operation, e);
    if (operation == DML_INSERT)
        callInsertProcedure(e);
    else if (operation == DML_UPDATE)
        callUpdateProcedure(e);
    else if (operation == DML_DELETE)
        callDeleteProcedure(e);
}
```

In the `PLSQUEntityImpl.java` base class, you can write the helper methods so that they perform the default processing like this:

```
// In PLSQUEntityImpl.java
/* Override in a subclass to perform non-default processing */
protected void callInsertProcedure(TransactionEvent e) {
    super.doDML(DML_INSERT, e);
}
/* Override in a subclass to perform non-default processing */
protected void callUpdateProcedure(TransactionEvent e) {
    super.doDML(DML_UPDATE, e);
}
/* Override in a subclass to perform non-default processing */
protected void callDeleteProcedure(TransactionEvent e) {
    super.doDML(DML_DELETE, e);
}
```

After putting this infrastructure in place, when you base an entity object on the `PLSQUEntityImpl` class, you can use the **Source | Override Methods** menu item to override the `callInsertProcedure()`, `callUpdateProcedure()`, and `callDeleteProcedure()` helper methods and perform the appropriate stored procedure calls for that particular entity. To simplify the task of implementing these calls, you could add the `call.StoredProcedure()` helper method you learned about in [Chapter 33.5, "Invoking Stored Procedures and Functions"](#) to the `PLSQUEntityImpl` class as well. This way, any PL/SQL-based entity objects that extend this class can leverage the helper method.

34.5.4 Implementing the Stored Procedure Calls for DML Operations

To implement the stored procedure calls for DML operations, do the following:

- Use the **Class Extends** button on the **Java** panel of the Overview Editor to set your `Products` entity object to have the `PLSQUEntityImpl` class as its base class.
- Enable a custom Java class for the `Products` entity object.
- Use the **Source | Override Methods** menu item and select the `callInsertProcedure()`, `callUpdateProcedure()`, and `callDeleteProcedure()` methods.

[Example 34–10](#) shows the code you would write in these overridden helper methods.

Example 34–10 Leveraging a Helper Method to Invoke Insert, Update, and Delete Procedures

```
// In ProductsImpl.java
protected void callInsertProcedure(TransactionEvent e) {
    callStoredProcedure("products_api.insert_product(?, ?, ?, ?)",
        new Object[] { getProductId(), getProductName(),
                      getSupplierId(), getListPrice(),
                      getMinPrice(), getShippingClassCode() });
}
protected void callUpdateProcedure(TransactionEvent e) {
    callStoredProcedure("products_api.update_product(?, ?, ?, ?)",
        new Object[] { getProductId(), getProductName(),
                      getSupplierId(), getListPrice(),
                      getMinPrice(), getShippingClassCode() });
}
protected void callDeleteProcedure(TransactionEvent e) {
    callStoredProcedure("products_api.delete_product(?)",
        new Object[] { getProductId() });
}
```

At this point, if you create a default entity-based view object called `Products` for the `Products` entity object and add an instance of it to a `ProductsModule` application module you can quickly test inserting, updating, and deleting rows from the `Products` view object instance in the Business Component Browser.

Often, overriding just the insert, update, and delete operations will be enough. The default behavior that performs the `SELECT` statement for `findByPrimaryKey()` and the `SELECT FOR UPDATE` statement for the `lock()` against the database view works for most basic kinds of views.

However, if the view is complex and does not support `SELECT FOR UPDATE` or if you need to perform the `findByPrimaryKey()` and `lock()` functionality using additional stored procedures API's, then you can follow the steps in the next section.

34.5.5 Adding Select and Lock Handling

You can also handle the lock and `findByPrimaryKey()` functionality of an entity object by invoking stored procedures if necessary. Imagine that the `PRODUCTS_API` package were updated to contain the two additional procedures shown in [Example 34–11](#). Both the `lock_product` and `select_product` procedures accept a primary key attribute as an `IN` parameter and return values for the remaining attributes using `OUT` parameters.

Example 34–11 Additional Locking and Select Procedures for the PRODUCTS Table

```
/* Added to PRODUCTS_API package */
procedure lock_product(p_prod_id number,
    p_name OUT varchar2,
    p_image OUT varchar2,
    p_description OUT varchar2);
procedure select_product(p_prod_id number,
    p_name OUT varchar2,
    p_image OUT varchar2,
    p_description OUT varchar2);
```

34.5.5.1 Updating PLSQLEntityImpl Base Class to Handle Lock and Select

You can extend the `PLSQLEntityImpl` base class to handle the `lock()` and `findByPrimaryKey()` overrides using helper methods similar to the ones you

added for insert, update, delete. At runtime, both the `lock()` and `findByPrimaryKey()` operations end up invoking the lower-level entity object method called `doSelect(boolean lock)`. The `lock()` operation calls `doSelect()` with a `true` value for the parameter, while the `findByPrimaryKey()` operation calls it passing `false` instead.

[Example 34–12](#) shows the overridden `doSelect()` method in `PLSLEntityImpl` to delegate as appropriate to two helper methods that subclasses can override as necessary.

Example 34–12 Overriding `doSelect()` to Call Different Procedures Based on the Lock Parameter

```
// In PLSLEntityImpl.java
protected void doSelect(boolean lock) {
    if (lock) {
        callLockProcedureAndCheckForRowInconsistency();
    } else {
        callSelectProcedure();
    }
}
```

The two helper methods are written to just perform the default functionality in the base `PLSLEntityImpl` class:

```
// In PLSLEntityImpl.java
/* Override in a subclass to perform non-default processing */
protected void callLockProcedureAndCheckForRowInconsistency() {
    super.doSelect(true);
}
/* Override in a subclass to perform non-default processing */
protected void callSelectProcedure() {
    super.doSelect(false);
}
```

Notice that the helper method that performs locking has the name `callLockProcedureAndCheckForRowInconsistency()`. This reminds developers that it is their responsibility to perform a check to detect at the time of locking the row whether the newly-selected row values are the same as the ones the entity object in the entity cache believes are the current database values.

To assist subclasses in performing this old-value versus new-value attribute comparison, you can add one final helper method to the `PLSLEntityImpl` class like this:

```
// In PLSLEntityImpl
protected void compareOldAttrTo(int attrIndex, Object newVal) {
    if ((getPostedAttribute(attrIndex) == null && newVal != null) ||
        (getPostedAttribute(attrIndex) != null && newVal == null) ||
        (getPostedAttribute(attrIndex) != null && newVal != null &&
           !getPostedAttribute(attrIndex).equals(newVal))) {
        throw new RowInconsistentException(getKey());
    }
}
```

34.5.5.2 Implementing Lock and Select for the Product Entity

With the additional infrastructure in place in the base `PLSLEntityImpl` class, you can override the `callSelectProcedure()` and `callLockProcedureAndCheckForRowInconsistency()` helper methods in the

Product entity object's `ProductImpl` class. Since the `select_product` and `lock_product` procedures have OUT arguments, as you learned in [Section 33.5.4, "Calling Other Types of Stored Procedures"](#), you need to use a JDBC `CallableStatement` object to perform these invocations.

[Example 34–13](#) shows the code required to invoke the `select_product` procedure. It's performing the following basic steps:

1. Creating a `CallableStatement` for the PLSQL block to invoke.
2. Registering the OUT parameters and types, by one-based bind variable position.
3. Setting the IN parameter value.
4. Executing the statement.
5. Retrieving the possibly updated column values.
6. Populating the possibly updated attribute values in the row.
7. Closing the statement.

Example 34–13 Invoking the Stored Procedure to Select a Row by Primary Key

```
// In ProductsImpl.java
protected void callSelectProcedure() {
    String stmt = "begin products_api.select_product(?, ?, ?, ?, ?, ?);end;";
    // 1. Create a CallableStatement for the PLSQL block to invoke
    CallableStatement st =
        getDBTransaction().createCallableStatement(stmt, 0);
    try {
        // 2. Register the OUT parameters and types
        st.registerOutParameter(2, VARCHAR2);
        st.registerOutParameter(3, NUMBER);
        st.registerOutParameter(4, NUMBER);
        st.registerOutParameter(5, NUMBER);
        st.registerOutParameter(6, VARCHAR2);

        // 3. Set the IN parameter value
        st.setObject(1, getProductId());
        // 4. Execute the statement
        st.executeUpdate();
        // 5. Retrieve the possibly updated column values
        String possiblyUpdatedName = st.getString(2);
        String possiblyUpdatedSupplierId = st.getString(3);
        String possiblyUpdatedListPrice = st.getString(4);
        String possiblyUpdatedMinPrice = st.getString(5);
        String possiblyUpdatedShipCode = st.getString(6);

        // 6. Populate the possibly updated attribute values in the row
        populateAttribute(PRODUCTNAME, possiblyUpdatedName, true, false,
                           false);
        populateAttribute(SUPPLIERID, possiblyUpdatedSupplierId, true,
                           false, false);
        populateAttribute(LISTPRICE, possiblyUpdatedListPrice, true, false,
                           false);
        populateAttribute(MINPRICE, possiblyUpdatedMinPrice, true, false,
                           false);
        populateAttribute(SHIPPINGCLASSCODE, possiblyUpdatedShipCode, true,
                           false, false);
    } catch (SQLException e) {
        throw new JboException(e);
    } finally {
```

```
        if (st != null) {
            try {
                // 7. Closing the statement
                st.close();
            } catch (SQLException e) {
            }
        }
    }
}
```

Example 34–14 shows the code to invoke the `lock_product` procedure. It's doing basically the same steps as above, with just the following two interesting differences:

- After retrieving the possibly updated column values from the `OUT` parameters, it uses the `compareOldAttrTo()` helper method inherited from the `PLSQUEntityImpl` to detect whether or not a `RowInconsistentException` should be thrown as a result of the row lock attempt.
- In the `catch (SQLException e)` block, it is testing to see whether the database has thrown the error:

```
ORA-00054: resource busy and acquire with NOWAIT specified
```

and if so, it again throws the ADF Business Components `AlreadyLockedException` just as the default entity object implementation of the `lock()` functionality would do in this situation.

Example 34–14 Invoking the Stored Procedure to Lock a Row by Primary Key

```
// In ProductsImpl.java
protected void callLockProcedureAndCheckForRowInconsistency() {
    String stmt = "begin products_api.lock_product(?, ?, ?, ?, ?, ?);end;";
    CallableStatement st =
        getDBTransaction().createCallableStatement(stmt, 0);
    try {
        st.registerOutParameter(2, VARCHAR2);
        st.registerOutParameter(3, NUMBER);
        st.registerOutParameter(4, NUMBER);
        st.registerOutParameter(5, NUMBER);
        st.registerOutParameter(6, VARCHAR2);
        st.setObject(1, getProductId());
        st.executeUpdate();
        String possiblyUpdatedName = st.getString(2);
        String possiblyUpdatedSupplierId = st.getString(3);
        String possiblyUpdatedListPrice = st.getString(4);
        String possiblyUpdatedMinPrice = st.getString(5);
        String possiblyUpdatedShipCode = st.getString(6);
        compareOldAttrTo(PRODUCTNAME, possiblyUpdatedName);
        compareOldAttrTo(SUPPLIERID, possiblyUpdatedSupplierId);
        compareOldAttrTo(LISTPRICE, possiblyUpdatedListPrice);
        compareOldAttrTo(MINPRICE, possiblyUpdatedMinPrice);
        compareOldAttrTo(SHIPPINGCLASSCODE, possiblyUpdatedShipCode);
    } catch (SQLException e) {
        if (Math.abs(e.getErrorCode()) == 54) {
            throw new AlreadyLockedException(e);
        } else {
            throw new JboException(e);
        }
    } finally {
        if (st != null) {
            try {
```

```
        st.close();
    } catch (SQLException e) {
        }
    }
}
```

With these methods in place, you have a `Products` entity object that wraps the `PRODUCTS_API` package for all of its database operations. Due to the clean separation of the data querying functionality of view objects and the data validation and saving functionality of entity objects, you can now leverage this `Products` entity object in any way you would use a normal entity object. You can build as many different view objects that use `Products` as their entity usage as necessary.

34.5.5.3 Refreshing the Entity Object After RowInconsistentException

You can override the `lock()` method to refresh the entity object after a `RowInconsistentException` has occurred. [Example 34-15](#) shows code that can be added to the entity object implementation class to catch the `RowInconsistentException` and refresh the entity object.

Example 34-15 Overridden lock() Method to Refresh Entity Object on RowInconsistentException

```
// In the entity object implementation class
@Override
public void lock() {
    try {
        super.lock();
    }
    catch (RowInconsistentException ex) {
        this.refresh(REFRESH_UNDO_CHANGES);
        throw ex;
    }
}
```

34.6 Basing an Entity Object on a Join View or Remote DBLink

If you need to create an entity object based on either of the following:

- Synonym that resolves to a remote table over a DBLINK
 - View with INSTEAD OF triggers

Then you will encounter the following error if any of its attributes are marked as **Refresh on Insert or Refresh on Update**:

```
JBO-26041: Failed to post data to database during "Update"  
## Detail 0 ##  
ORA-22816: unsupported feature with RETURNING clause
```

These types of schema objects do not support the RETURNING clause, which by default the entity object uses to more efficiently return the refreshed values in the same database roundtrip in which the INSERT or UPDATE operation was executed.

To disable the use of the RETURNING clause for an entity object of this type, do the following:

1. Enable a custom entity definition class for the entity object.

2. In the custom entity definition class, override the `createDef()` method to call:
`setUseReturningClause(false)`
3. If the **Refresh on Insert** attribute is the primary key of the entity object, you must identify some other attribute in the entity as an alternate unique key by setting the **Unique Key** property on it.

At runtime, when you have disabled the use of the RETURNING clause in this way, the entity object implements the **Refresh on Insert** and **Refresh on Update** behavior using a separate SELECT statement to retrieve the values to refresh after insert or update as appropriate.

34.7 Using Inheritance in Your Business Domain Layer

Inheritance is a powerful feature of object-oriented development that can simplify development and maintenance when used appropriately. As you've seen in [Section 33.9, "Creating Extended Components Using Inheritance"](#), ADF Business Components supports using inheritance to create new components that extend existing ones in order to add additional properties or behavior or modify the behavior of the parent component. This section helps you understand when inheritance can be useful in modeling the different kinds of entities in your reusable business domain layer.

Note: The examples in this section refer to the `InheritanceAndPolymorphicQueries` project in the `AdvancedEntityExamples` workspace of the Fusion Order Demo. See the note at the beginning of this chapter for download instructions. Run the `AlterPersonsTable.sql` script in the `src` folder against the FOD connection to set up the additional database objects required for the project.

34.7.1 Understanding When Inheritance Can be Useful

Your application's database schema might contain tables where different logical kinds of business information are stored in rows of the same table. These tables will typically have one column whose value determines the kind of information stored in each row. For example, the Fusion Order Demo's `PERSONS` table stores information about customers, suppliers, and staff in the same table. It contains a `PERSON_TYPE_CODE` column whose value — `STAFF`, `CUST`, or `SUPP` — determines what kind of `PERSON` the row represents.

While the Fusion Order Demo implementation doesn't yet contain this differentiation in this release, it's reasonable to assume that a future release of the application might require:

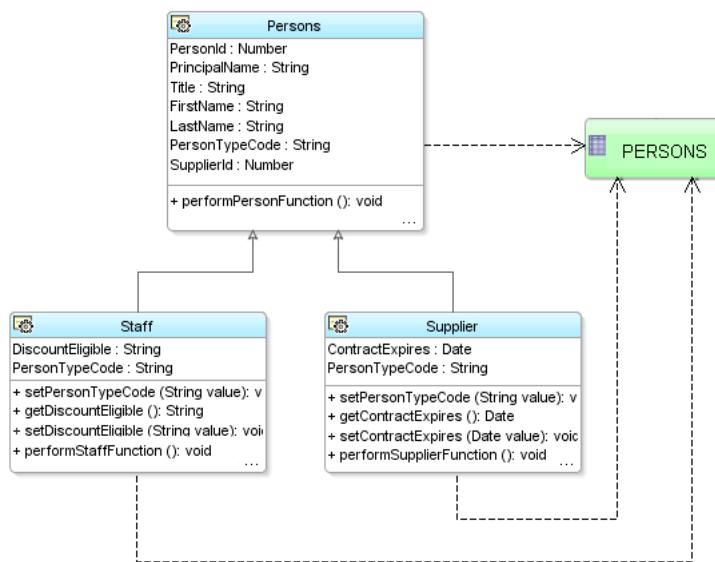
- Managing additional database-backed attributes that are specific to suppliers or specific to staff
- Implementing common behavior for all users that is different for suppliers or staff
- Implementing new functionality that is specific to only suppliers or only staff

[Figure 34-1](#) shows what the business domain layer would look like if you created distinct `Persons`, `Staff`, and `Supplier` entity objects to allow distinguishing the different kinds of business information in a more formal way inside your application. Since suppliers and staff are special *kinds* of persons, their corresponding entity objects would extend the base `Persons` entity object. This base `Persons` entity object

contains all of the attributes and methods that are common to all types of users. The `performPersonFunction()` method in the figure represents one of these common methods.

Then, for the `Supplier` and `Staff` entity objects you can add specific additional attributes and methods that are unique to that kind of user. For example, `Supplier` has an additional `ContractExpires` attribute of type `Date` to track when the supplier's current contract expires. There is also a `performSupplierFunction()` method that is specific to suppliers. Similarly, the `Staff` entity object has an additional `DiscountEligible` attribute to track whether the person qualifies for a staff discount. The `performStaffFunction()` is a method that is specific to staff.

Figure 34-1 Distinguishing Persons, Suppliers, and Staff Using Inheritance



By modeling these different kinds of persons as distinct entity objects in an inheritance hierarchy in your domain business layer, you can simplify having them share common data and behavior and implement the aspects of the application that make them distinct.

34.7.2 How To Create Entity Objects in an Inheritance Hierarchy

To create entity objects in an inheritance hierarchy, you use the Create Entity Object wizard to create each entity following the steps outlined in the sections below. The example described here assumes that you've altered the FOD application's `PERSONS` table by executing the following DDL statement to add two new columns to it:

```

alter table persons add (
    discount_eligible varchar2(1),
    contract_expires date
);

```

34.7.2.1 Start By Identifying the Discriminator Column and Distinct Values

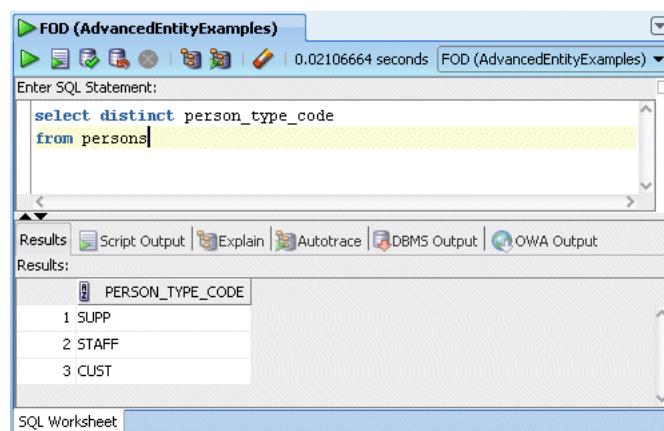
Before creating entity objects in an inheritance hierarchy based on table containing different kinds of information, you should first identify which column in the table is used to distinguish the kind of row it is. In the FOD application's `PERSONS` table, this is the `PERSON_TYPE_CODE` column. Since it helps partition or "discriminate" the rows in the table into separate groups, this column is known as the discriminator column.

Next, determine the valid values that the discriminator column takes on in your table. You might know this off the top of your head, or you could execute a simple SQL statement in the JDeveloper **SQL Worksheet** to determine the answer. To access the worksheet:

1. Choose **Database Navigator** from the **View** menu.
2. Expand the **AdvancedEntityExamples** folder and select the FOD connection.
3. Right-click FOD, and choose **Open SQL Worksheet** from the context menu.

Figure 34–2 shows the results of performing a `SELECT DISTINCT` query in the SQL Worksheet on the `PERSON_TYPE_CODE` column in the `PERSONS` table. It confirms that the rows are partitioned into three groups based on the `PERSON_TYPE_CODE` discriminator values: SUPP, STAFF, and CUST.

Figure 34–2 Using the SQL Worksheet to Find Distinct Discriminator Column Values



34.7.2.2 Identify the Subset of Attributes Relevant to Each Kind of Entity

Once you know how many different kinds of business entities are stored in the table, you will also know how many entity objects to create to model these distinct items. You'll typically create one entity object per kind of item. Next, in order to help determine which entity should act as the base of the hierarchy, you need to determine which subset of attributes is relevant to each kind of item.

Using the example above, assume you determine that all of the attributes except `ContractExpires` and `DiscountEligible` are relevant to all users, that `ContractExpires` is specific to suppliers, and that `DiscountEligible` is specific to staff. This information leads you to determine that the `Persons` entity object should be the base of the hierarchy, with the `Supplier` and `Staff` entity objects each extending `Persons` to add their specific attributes.

34.7.2.3 Creating the Base Entity Object in an Inheritance Hierarchy

To create the base entity object in an inheritance hierarchy, use the Create Entity Object wizard and follow these steps:

- In step 1 on the Name panel, provide a name and package for the entity, and select the schema object on which the entity will be based.
For example, name the entity object `Persons` and base it on the `PERSONS` table.
- In step 2 on the Attributes panel, select the attributes in the **Entity Attributes** list that are not relevant to the base entity object (if any) and click **Remove** to remove them.

For example, remove the `DiscountEligible` and `ContractExpires` attributes from the list.

- In step 3 on the Attribute Settings panel, use the **Select Attribute** dropdown list to choose the attribute that will act as the discriminator for the family of inherited entity objects and check the **Discriminator** checkbox to identify it as such. Importantly, you must also supply a **Default Value** for this discriminator attribute to identify rows of this base entity type.

For example, select the `PersonTypeCode` attribute, mark it as a discriminator attribute, and set its **Default Value** to the value "cust".

Note: Leaving the **Default Value** blank for a discriminator attribute is legal. A blank default value means that a row whose discriminator column value IS NULL will be treated as this base entity type.

Then click **Finish** to create the entity object.

34.7.2.4 Creating a Subtype Entity Object in an Inheritance Hierarchy

To create a subtype entity object in an inheritance hierarchy, first do the following:

- Determine the entity object that will be the parent entity object from which your new entity object will extend.

For example, the parent entity for a new `Manager` entity object will be the `User` entity created above.

- Ensure that the parent entity has a discriminator attribute already identified.

The base type must already have the discriminator attribute identified as described in the section above. If it does not, use the overview editor to set the **Discriminator** property on the appropriate attribute of the parent entity before creating the inherited child.

Then, use the Create Entity Object wizard and follow these steps to create the new subtype entity object in the hierarchy:

- In step 1 on the Name panel, provide a name and package for the entity, and click the **Browse** button next to the **Extends** field to select the parent entity from which the entity being created will extend.

For example, name the new entity `Staff` and select the `Persons` entity object in the **Extends** field.

- In step 2, the Attributes panel displays the attributes from the underlying table that are not included in the base entity object. Select the attributes you do not want to include in this entity object and click **Remove**.

For example, since you are creating the `Staff` entity remove the `ContractExpires` attribute and leave the `DiscountEligible` attribute.

- Still on step 2, click **Override** to select the discriminator attribute so that you can customize the attribute metadata to supply a distinct **Default Value** for the `Staff` subtype.

For example, override the `PersonTypeCode` attribute.

- In step 3 on the Attribute Settings panel, use the **Select Attribute** dropdown list to select the discriminator attribute. Change the **Default Value** field to supply a

distinct default value for the discriminator attribute that defines the entity subtype being created.

For example, select the `PersonTypeCode` attribute and change its **Default Value** to the value "staff".

Then click **Finish** to create the subtype entity object.

Note: You can repeat the same steps to define the `Supplier` entity object that extends `Persons` to add the additional `ContractExpires` attribute and overrides the **Default Value** of the `UserRole` discriminator attribute to have the value "supp".

34.7.3 How to Add Methods to Entity Objects in an Inheritance Hierarchy

To add methods to entity objects in an inheritance hierarchy, enable the custom Java class for the entity object and visit the code editor to add the method.

34.7.3.1 Adding Methods Common to All Entity Objects in the Hierarchy

To add a method that is common to all entity objects in the hierarchy, enable a custom Java class for the base entity object in the hierarchy and add the method in the code editor. For example, if you add the following method to the `PersonsImpl` class for the base `User` entity object, it will be inherited by all entity objects in the hierarchy:

```
// In PersonsImpl.java
public void performPersonFunction() {
    System.out.println("## performPersonFunction as Customer");
}
```

34.7.3.2 Overriding Common Methods in a Subtype Entity

To override a method in a subtype entity that is common to all entity objects in the hierarchy, enable a custom Java class for the subtype entity and choose **Override Methods** from the **Source** menu to launch the Override Methods dialog. Select the method you want to override, and click **OK**. Then, customize the overridden method's implementation in the code editor. For example, imagine overriding the `performPersonFunction()` method in the `StaffImpl` class for the `Staff` subtype entity object and change the implementation to look like this:

```
// In StaffImpl.java
public void performPersonFunction() {
    System.out.println("## performPersonFunction as Staff");
}
```

When working with instances of entity objects in a subtype hierarchy, sometimes you will process instances of multiple different subtypes. Since `Staff` and `Supplier` entities are special kinds of `Persons`, you can write code that works with all of them using the more generic `PersonsImpl` type that they all have in common. When doing this generic kind of processing of classes that might be one of a *family* of subtypes in a hierarchy, Java will always invoke the most specific override of a method available.

This means that invoking the `performPersonFunction()` method on an instance of `PersonsImpl` that happens to really be the more specific `StaffImpl` subtype, will the result in printing out the following:

```
## performPersonFunction as Staff
```

instead of the default result that regular `PersonsImpl` instances would get:

```
## performPersonFunction as Customer
```

34.7.3.3 Adding Methods Specific to a Subtype Entity

To add a method that is specific to a subtype entity object in the hierarchy, enable a custom Java class for that entity and add the method in the code editor. For example, you could add the following method to the `SupplierImpl` class for the `Supplier` subtype entity object:

```
// In SupplierImpl.java
public void performSupplierFunction() {
    System.out.println("## performSupplierFunction called");
}
```

34.7.4 What You May Need to Know About Using Inheritance

The sections below describe some of the things you may need to know about when using inheritance.

34.7.4.1 Sometimes You Need to Introduce a New Base Entity

In the example above, the `Persons` entity object corresponded to a concrete kind of row in the `PERSONS` table and it also played the role of the base entity in the hierarchy. In other words, all of its attributes were common to all entity objects in the hierarchy. You might wonder what would happen, however, if the `Persons` entity required a property that was specific to customers, but not common to staff or suppliers. Imagine that customers can participate in customer satisfaction surveys, but that staff and suppliers do not. The `Persons` entity would require a `LastSurveyDate` attribute to handle this requirement, but it wouldn't make sense to have `Staff` and `Supplier` entity objects inherit it.

In this case, you can introduce a new entity object called `BasePersons` to act as the base entity in the hierarchy. It would have all of the attributes common to all `Persons`, `Staff`, and `Supplier` entity objects. Then each of the three entities the correspond to concrete rows that appear in the table could have some attributes that are inherited from `BasePersons` and some that are specific to its subtype. In the `BasePersons` type, so long as you mark the `PersonTypeCode` attribute as a discriminator attribute, you can just leave the **Default Value** blank (or some other value that does *not* occur in the `PERSON_TYPE_CODE` column in the table). Because at runtime you'll never be using instances of the `BasePersons` entity, it doesn't really matter what its discriminator default value is.

34.7.4.2 Finding Subtype Entities by Primary Key

When you use the `findByPrimaryKey()` method on an entity definition, it only searches the entity cache for the entity object type on which you call it. In the example above, this means that if you call `PersonsImpl.getDefinitionObject()` to access the entity definition for the `Persons` entity object when you call `findByPrimaryKey()` on it, you will only find entities in the cache that happen to be customers. Sometimes this is exactly the behavior you want. However, if you want to find an entity object by primary key allowing the possibility that it might be a subtype in an inheritance hierarchy, then you can use the `EntityDefImpl` class' `findByPKExtended()` method instead. In the `Persons` example described here, this alternative finder method would find an entity object by primary key whether it is a customer, supplier, or staff. You can then use the Java `instanceof` operator to test which type you found, and then cast the `PersonsImpl` object to the more specific entity object type in order to work with features specific to that subtype.

34.7.4.3 You Can Create View Objects with Polymorphic Entity Usages

When you create an entity-based view object with an entity usage corresponding to a base entity object in an inheritance hierarchy, you can configure the view object to query rows corresponding to multiple different subtypes in the base entity's subtype hierarchy. Each row in the view object will use the appropriate subtype entity object as the entity row part, based on matching the value of the discriminator attribute. See [Section 35.7.2, "How To Create a View Object with a Polymorphic Entity Usage"](#) for specific instructions on setting up and using these view objects.

34.8 Controlling Entity Posting Order to Avoid Constraint Violations

Due to database constraints, when you perform DML operations to save changes to a number of related entity objects in the same transaction, the order in which the operations are performed can be significant. If you try to insert a new row containing foreign key references *before* inserting the row being referenced, the database can complain with a constraint violation. This section helps you understand the default order for processing of entity objects during commit time and how to programmatically influence that order when necessary.

Note: The examples in this section refer to the `ControllingPostingOrder` project in the `AdvancedEntityExamples` workspace. See the note at the beginning of this chapter for download instructions.

34.8.1 Understanding the Default Post Processing Order

By default, when you commit the transaction the entity objects in the pending changes list are processed in chronological order, in other words, the order in which the entities were added to the list. This means that, for example, if you create a new `Product` and then a new `Supplier` related to that product, the new `Product` will be inserted first and the new `Supplier` second.

34.8.2 How Compositions Change the Default Processing Ordering

When two entity objects are related by a *composition*, the strict chronological ordering is modified automatically to ensure that composed parent and child entity rows are saved in an order that avoids violating any constraints. This means, for example, that a new parent entity row is inserted before any new composed children entity rows.

34.8.3 Overriding `postChanges()` to Control Post Order

If your related entities are associated but *not* composed, then you need to write a bit of code to ensure that the related entities get saved in the appropriate order.

34.8.3.1 Observing the Post Ordering Problem First Hand

Consider the `newProductForNewSupplier()` custom method from an `PostModule` application module in [Example 34-16](#). It accepts a set of parameters and:

1. Creates a new `Product`.
2. Creates a new `Supplier`.
3. Sets the product id to which the server request pertains.
4. Commits the transaction.

5. Constructs a Result Java bean to hold new product ID and supplier ID.
6. Results the result.

Note: The code makes the assumption that both Products.ProductId and Suppliers.SupplierId have been set to have DBSequence data type to populate their primary keys based on a sequence.

Example 34–16 Creating a New Product Then a New Supplier and Returning the New Ids

```
// In PostModuleImpl.java
public Result newProductForNewSupplier(String supplierName,
                                         String supplierStatus,
                                         String productName,
                                         String productStatus,
                                         Number listPrice,
                                         Number minPrice,
                                         String shipCode) {
    oracle.jbo.domain.Date today = new Date(Date.getCurrentDate());
    Number objectId = new Number(0);
    // 1. Create a new product
    ProductsBaseImpl newProduct = createNewProduct();
    // 2. Create a new supplier
    SuppliersImpl newSupplier = createNewSupplier();
    newSupplier.setSupplierName(supplierName);
    newSupplier.setSupplierStatus(supplierStatus);
    newSupplier.setCreatedBy("PostingModule");
    newSupplier.setCreationDate(today);
    newSupplier.setLastUpdatedBy("PostingModule");
    newSupplier.setLastUpdateDate(today);
    newSupplier.setObjectVersionId(objectId);
    // 3. Set the supplier id to which the product pertains
    newProduct.setSupplierId(newSupplier.getSupplierId().getSequenceNumber());
    newProduct.setProductName(productName);
    newProduct.setProductStatus(productStatus);
    newProduct.setListPrice(listPrice);
    newProduct.setMinPrice(minPrice);
    newProduct.setShippingClassCode(shipCode);
    newProduct.setCreatedBy("PostingModule");
    newProduct.setCreationDate(today);
    newProduct.setLastUpdatedBy("PostingModule");
    newProduct.setLastUpdateDate(today);
    newProduct.setObjectVersionId(objectId);
    // 4. Commit the transaction
    getDBTransaction().commit();
    // 5. Construct a bean to hold new supplier id and product id
    Result result = new Result();
    result.setProductId(newProduct.getProductId().getSequenceNumber());
    result.setSupplierId(newSupplier.getSupplierId().getSequenceNumber());
    // 6. Return the result
    return result;
}
private ProductsBaseImpl createNewProduct(){
    EntityDefImpl productDef = ProductsBaseImpl.getDefinitionObject();
    return (ProductsBaseImpl) productDef.createInstance2(getDBTransaction(), null);
}

private SuppliersImpl createNewSupplier(){
```

```

EntityDefImpl supplierDef = SuppliersImpl.getDefinitionObject();
return (SuppliersImpl) supplierDef.createInstance2(getDBTransaction(), null);
}

```

If you add this method to the application module's client interface and test it from a test client program, you get an error:

```

oracle.jbo.DMLConstraintException:
JBO-26048: Constraint "PRODUCT_SUPPLIER_FK" violated during post operation:
"Insert" using SQL Statement
"BEGIN
  INSERT INTO PRODUCTS(
    SUPPLIER_NAME, SUPPLIER_STATUS, PRODUCT_NAME,
    PRODUCT_STATUS, LIST_PRICE, MIN_PRICE, SHIPPING_CLASS_CODE)
  VALUES (?,?,?,?,?, ?, ?)
  RETURNING PRODUCT_ID INTO ?;
END; ".
## Detail 0 ##
java.sql.SQLException:
ORA-02291: integrity constraint (FOD.PRODUCT_SUPPLIER_FK) violated
- parent key not found

```

The database complains when the PRODUCTS row is inserted that the value of its SUPPLIER_ID foreign key doesn't correspond to any row in the SUPPLIERS table. This occurred because:

- The code created the Product before the Supplier
- Products and Suppliers entity objects are associated but not composed
- The DML operations to save the new entity rows is done in chronological order, so the new Product gets inserted before the new Supplier.

34.8.3.2 Forcing the Supplier to Post Before the Product

To remedy the problem, you could reorder the lines of code in the example to create the Supplier first, then the Product. While this would address the immediate problem, it still leaves the chance that another application developer could creating things in an incorrect order.

The better solution is to make the entity objects *themselves* handle the posting order so it will work correctly regardless of the order of creation. To do this you need to override the `postChanges()` method in the entity that contains the foreign key attribute referencing the associated entity object and write code as shown in [Example 34-17](#). In this example, since it is the Product that contains the foreign key to the Supplier entity, you need to update the Product to conditionally force a related, new Supplier to post before the service request posts itself.

The code tests whether the entity being posted is in the STATUS_NEW or STATUS_MODIFIED state. If it is, it retrieves the related product using the `getSupplier()` association accessor. If the related Supplier also has a post-state of STATUS_NEW, then *first* it calls `postChanges()` on the related parent row before calling `super.postChanges()` to perform its own DML.

Example 34-17 Overriding `postChanges()` in `ProductsBaseImpl` to Post Supplier First

```

// In ProductsBaseImpl.java
public void postChanges(TransactionEvent e) {
    /* If current entity is new or modified */
    if (getPostState() == STATUS_NEW ||
        getPostState() == STATUS_MODIFIED) {

```

```

/* Get the associated supplier for the product */
SuppliersImpl supplier = getSupplier();
/* If there is an associated supplier */
if (supplier != null) {
    /* And if it's post-status is NEW */
    if (supplier.getPostState() == STATUS_NEW) {
        /*
         * Post the supplier first, before posting this
         * entity by calling super below
         */
        supplier.postChanges(e);
    }
}
super.postChanges(e);
}

```

If you were to re-run the example now, you would see that without changing the creation order in the `newProductForNewSupplier()` method's code, entities now post in the correct order — first new Supplier, then new Product. Yet, there is still a problem. The constraint violation still appears, but now for a different reason!

If the primary key for the `Suppliers` entity object were user-assigned, then the code in [Example 34-17](#) would be all that is required to address the constraint violation by correcting the post ordering.

Note: An alternative to the programmatic technique discussed above, which solves the problem at the Java EE application layer, is the use of deferrable constraints at the database layer. If you have control over your database schema, consider defining (or altering) your foreign key constraints to be DEFERRABLE INITIALLY DEFERRED. This causes the database to defer checking the constraint until transaction commit time. This allows the application to perform DML operations in any order provided that by COMMIT time all appropriate related rows have been saved and would alleviate the parent/child ordering described above. However, you would still need to write the code described in the following sections to cascade-update the foreign key values if the parent's primary key is assigned from a sequence.

In this example, however, the `Suppliers.SupplierId` is assigned from a database sequence, and not user-assigned in this example. So when a new `Suppliers` entity row gets posted its `SupplierId` attribute is refreshed to reflect the database-assigned sequence value. The foreign key value in the `Products.SupplierId` attribute referencing the new supplier is "orphaned" by this refreshing of the supplier's ID value. When the product's row is saved, its `SUPPLIER_ID` value still doesn't match a row in the `SUPPLIERS` table, and the constraint violation occurs again. The next two sections discuss the solution to address this "orphaning" problem.

34.8.3.3 Understanding Associations Based on DBSequence-Valued Primary Keys

Recall from [Section 4.10.10, "How to Get Trigger-Assigned Primary Key Values from a Database Sequence"](#) that when an entity object's primary key attribute is of `DBSequence` type, during the transaction in which it is created, its numerical value is a unique, temporary negative number. If you create a number of associated entities in the same transaction, the relationships between them are based on this temporary

negative key value. When the entity objects with DBSequence-value primary keys are posted, their primary key is refreshed to reflect the correct database-assigned sequence number, leaving the associated entities that are still holding onto the temporary negative foreign key value "orphaned".

For entity objects based on a *composition*, when the parent entity object's DBSequence-valued primary key is refreshed, the composed children entity rows automatically have their temporary negative foreign key value updated to reflect the owning parent's refreshed, database-assigned primary key. This means that for composed entities, the "orphaning" problem does not occur.

However, when entity objects are related by an association that is not a composition, you need to write a little code to insure that related entity rows referencing the temporary negative number get updated to have the refreshed, database-assigned primary key value. The next section outlines the code required.

34.8.3.4 Refreshing References to DBSequence-Assigned Foreign Keys

When an entity like `Suppliers` in this example has a DBSequence-valued primary key, and it is referenced as a foreign key by other entities that are associated with (but not composed by) it, you need to override the `postChanges()` method as shown in [Example 34-18](#) to save a reference to the row set of entity rows that might be referencing this new `Suppliers` row. If the status of the current `Suppliers` row is `New`, then the code assigns the `RowSet`-valued return of the `getProduct()` association accessor to the `newProductsBeforePost` member field before calling `super.postChanges()`.

Example 34-18 Saving Reference to Entity Rows Referencing this New Supplier

```
// In SuppliersImpl.java
RowSet newProductsBeforePost = null;
public void postChanges(TransactionEvent TransactionEvent) {
    /* Only bother to update references if Product is a NEW one */
    if (getPostState() == STATUS_NEW) {
        /*
         * Get a rowset of products related
         * to this new supplier before calling super
         */
        newProductsBeforePost = (RowSet)getProductsBase();
    }
    super.postChanges(TransactionEvent);
}
```

This saved `RowSet` is then used by the overridden `refreshFKInNewContainees()` method shown in [Example 34-19](#). It gets called to allow a new entity row to cascade update its refreshed primary key value to any other entity rows that were referencing it before the call to `postChanges()`. It iterates over the `ProductsBaseImpl` rows in the `newProductsBaseBeforePost` row set (if non-null) and sets the new supplier ID value of each one to the new sequence-assigned supplier value of the newly-posted `Suppliers` entity.

Example 34-19 Cascade Updating Referencing Entity Rows with New SupplierId Value

```
// In SuppliersImpl.java
protected void refreshFKInNewContainees() {
    if (newProductsBeforePost != null) {
        Number newSupplierId = getSupplierId().getSequenceNumber();
        /*
         * Process the rowset of products that referenced
         */
```

```

        * the new supplier prior to posting, and update their
        * SupplierId attribute to reflect the refreshed SupplierId value
        * that was assigned by a database sequence during posting.
    */
    while (newProductsBeforePost.hasNext()) {
        ProductsBaseImpl svrReq =
            (ProductsBaseImpl)newProductsBeforePost.next();
        product.setSupplierId(newSupplierId);
    }
    closeNewProductRowSet();
}
}

```

After implementing this change, the code in [Example 34–16](#) runs without encountering any database constraint violations.

34.9 Implementing Automatic Attribute Recalculation

[Section 4.13, "Adding Transient and Calculated Attributes to an Entity Object"](#) explained how to add calculated attributes to an entity object. Often the formula for the calculated value will depend on other attribute values in the entity. For example, consider a `LineItem` entity object representing the line item of an order. The `LineItem` might have attributes like `Price` and `Quantity`. You might introduce a calculated attributed named `ExtendedTotal` which you calculate by multiplying the price times the quantity. When either the `Price` or `Quantity` attributes is modified, you might expect the calculated attribute `ExtendedTotal` to be updated to reflect the new extended total, but this does not happen automatically. Unlike a spreadsheet, the entity object does not have any built-in expression evaluation engine that understands what attributes your formula depends on.

To address this limitation, you can write code in a framework extension class for entity objects that adds a recalculation facility. The code shown in [Example 34–20](#) demonstrates how to implement this. This code does not implement a sophisticated expression evaluator. Instead, it leverages the custom properties mechanism to allow a developer to supply a declarative "hint" about which attributes (for example, `X`, `Y`, and `Z`) should be recalculated when another attribute like `A` gets changed.

To leverage the generic facility, when implementing an entity object, you can:

- Base the entity on the framework extension class containing this additional code.
- Define one or more entity-level custom properties that follow a particular naming pattern. These indicate to the generic code which attributes should get recalculated when another specified attribute changes.

To indicate that "when attribute `A` changes, recalculate attributes `X`, `Y`, and `Z`," add a custom property named `Recalc_A` with the comma-separated value "`X, Y, Z`" to indicate that.

To implement this functionality the entity framework extension class in the example overrides the `notifyAttributesChanged()` method. This method gets invoked whenever the value of entity object attributes change. As arguments, the method receives two arrays:

- `int []` containing the index numbers of attributes whose values have changed.
- `Object []` containing the original values of the changed attributes. To access the new values, you can use the `getAttribute()` method or the typesafe attribute-getter for the attribute in question.

The code does the following basic steps:

1. Iterates over the set of custom entity properties.
2. If property name starts with "Recalc_" it gets the substring following this prefix to know the name of the attribute whose change should trigger recalculation of others.
3. Determines the index of the recalc-triggering attribute.
4. If the array of changed attribute indexes includes the index of the recalc-triggering attribute, then tokenize the comma-separated value of the property to find the names of the attributes to recalculate.
5. If there were any attributes to recalculate, add their attribute indexes to a new int [] of attributes whose values have changed.

The new array is created by copying the existing array elements in the attrIndices array to a new array, then adding in the additional attribute index numbers.

6. Call the super with the possibly updated array of changed attributes.

Example 34–20 Entity Framework Extension Code to Automatically Recalculate Derived Attributes

```
// In your entity framework extension class
protected void notifyAttributesChanged(int[] attrIndices, Object[] values) {
    int attrIndexCount = attrIndices.length;
    EntityDefImpl def = getEntityDef();
    HashMap eoProps = def.getPropertiesMap();
    if (eoProps != null && eoProps.size() > 0) {
        Iterator iter = eoProps.keySet().iterator();
        ArrayList otherAttrIndices = null;
        // 1. Iterate over the set of custom entity properties
        while (iter.hasNext()) {
            String curPropName = (String)iter.next();
            if (curPropName.startsWith(RECALC_PREFIX)) {
                // 2. If property name starts with "Recalc_" get follow attr name
                String changingAttrNameToCheck = curPropName.substring(PREFIX_LENGTH);
                // 3. Get the index of the recalc-triggering attribute
                int changingAttrIndexToCheck =
                    def.findAttributeDef(changingAttrNameToCheck).getIndex();
                if (isAttrIndexInList(changingAttrIndexToCheck, attrIndices)) {
                    // 4. If list of changed attrs includes recalc-triggering attr,
                    // then tokenize the comma-separated value of the property
                    // to find the names of the attributes to recalculate
                    String curPropValue = (String)eoProps.get(curPropName);
                    StringTokenizer st = new StringTokenizer(curPropValue, ",");
                    if (otherAttrIndices == null) {
                        otherAttrIndices = new ArrayList();
                    }
                    while (st.hasMoreTokens()) {
                        String attrName = st.nextToken();
                        int attrIndex = def.findAttributeDef(attrName).getIndex();
                        if (!isAttrIndexInList(attrIndex, attrIndices)) {
                            Integer intAttr = new Integer(attrIndex);
                            if (!otherAttrIndices.contains(intAttr)) {
                                otherAttrIndices.add(intAttr);
                            }
                        }
                    }
                }
            }
        }
    }
}
```

```

if (otherAttrIndices != null && otherAttrIndices.size() > 0) {
    // 5. If there were any attributes to recalculate, add their attribute
    // indexes to the int[] of attributes whose values have changed
    int extraAttrsToAdd = otherAttrIndices.size();
    int[] newAttrIndices = new int[attrIndexCount + extraAttrsToAdd];
    Object[] newValues = new Object[attrIndexCount + extraAttrsToAdd];
    System.arraycopy(attrIndices, 0, newAttrIndices, 0, attrIndexCount);
    System.arraycopy(values, 0, newValues, 0, attrIndexCount);
    for (int z = 0; z < extraAttrsToAdd; z++) {
        newAttrIndices[attrIndexCount+z] =
            ((Integer)otherAttrIndices.get(z)).intValue();
        newValues[attrIndexCount+z] =
            getAttribute((Integer)otherAttrIndices.get(z));
    }
    attrIndices = newAttrIndices;
    values = newValues;
}
}
// 6. Call the super with the possibly updated array of changed attributes
super.notifyAttributesChanged(attrIndices, values);
}

```

After implementing this code, you could then use this feature by setting a custom entity property named Recalc_OrderStatus with the value of Hidden. This way, anytime the value of the OrderStatus attribute is changed, the value of the calculated Hidden attribute is recalculated.

34.10 Implementing Custom Validation Rules

ADF Business Components comes with a base set of built-in declarative validation rules that you can use. However, a powerful feature of the validator architecture for entity objects is that you can create your own custom validation rules. When you notice that you or your team are writing the same kind of validation code over and over, you can build a custom validation rule class that captures this common validation "pattern" in a parameterized way.

Once you've defined a custom validation rule class, you can register it in JDeveloper so that it is as simple to use as any of the built-in rules. In fact, as you see in the following sections, you can even bundle your custom validation rule with a custom UI panel that JDeveloper leverages to facilitate developers' using and configuring the parameters your validation rule might require.

34.10.1 How To Create a Custom Validation Rule

To write a custom validation rule for entity objects, you need a Java class that implements the `JboValidatorInterface` in the `oracle.jbo.rules` package. You can create a skeleton class from the New Gallery.

To create a custom validator:

1. In the Application Navigator, right-click the project where you want to create the validator, and choose **New** from the context menu.
2. In the New Gallery, expand **ADF Business Components**, and then select **Validation Rule** and click **OK**.

As shown in [Example 34-21](#), `JBOValidatorInterface` contains one main `validate()` method, and a getter and setter method for a `Description` property.

Example 34–21 All Validation Rules Must Implement the JboValidatorInterface

```
package oracle.jbo.rules;
public interface JboValidatorInterface {
    void validate(JboValidatorContext valCtx) { }
    java.lang.String getDescription() { }
    void setDescription(String description) { }
}
```

If the behavior of your validation rule will be parameterized to make it more flexible, then add additional bean properties to your validator class for each parameter. For example, the code in [Example 34–22](#) implements a custom validation rule called DateMustComeAfterRule which validates that one date attribute must come after another date attribute. To allow the developer using the rule to configure the names of the date attributes to use as the initial and later dates for validation, this class defines two properties initialDateAttrName and laterDateAttrName.

[Example 34–22](#) shows the code that implements the custom validation rule. It extends the AbstractValidator to inherit support for working with the entity object's custom message bundle, where JDeveloper saves the validation error message when a developer uses the rule in an entity object.

The validate() method of the validation rule gets invoked at runtime whenever the rule class should perform its functionality. The code performs the following basic steps:

1. Ensures validator is correctly attached at the entity level.
2. Gets the entity row being validated.
3. Gets the values of the initial and later date attributes.
4. Validate that initial date is before later date.
5. Throws an exception if the validation fails.

Example 34–22 Custom DateMustComeAfterRule

```
// NOTE: package and imports omitted
public class DateMustComeAfterRule extends AbstractValidator
    implements JboValidatorInterface {
    /**
     * This method is invoked by the framework when the validator should do its job
     */
    public void validate(JboValidatorContext valCtx) {
        // 1. If validator is correctly attached at the entity level...
        if (validatorAttachedAtEntityLevel(valCtx)) {
            // 2. Get the entity row being validated
            EntityImpl eo = (EntityImpl)valCtx.getSource();
            // 3. Get the values of the initial and later date attributes
            Date initialValue = (Date) eo.getAttribute(getInitialDateAttrName());
            Date laterValue = (Date) eo.getAttribute(getLaterDateAttrName());
            // 4. Validate that initial date is before later date
            if (!validateValue(initialValue, laterValue)) {
                // 5. Throw the validation exception
                RulesBeanUtils.raiseException(getErrorMessageClass(),
                    getErrorMsgId(),
                    valCtx.getSource(),
                    valCtx.getSourceType(),
                    valCtx.getSourceFullName(),
                    valCtx.getAttributeDef(),
                    valCtx.getNewValue(),
                    null, null);
            }
        }
    }
}
```

```

        }
    }
    else {
        throw new RuntimeException("Rule must be at entity level");
    }
}
/***
 * Validate that the initialDate comes before the laterDate.
 */
private boolean validateValue(Date initialDate, Date laterDate) {
    return (initialDate == null) || (laterDate == null) ||
    (initialDate.compareTo(laterDate) < 0);
}
/***
 * Return true if validator is attached to entity object
 * level at runtime.
 */
private boolean validatorAttachedAtEntityLevel(JboValidatorContext ctx) {
    return ctx.getOldValue() instanceof EntityImpl;
}
// NOTE: Getter/Setter Methods omitted
private String description;
private String initialDateAttrName;
private String laterDateAttrName;
}

```

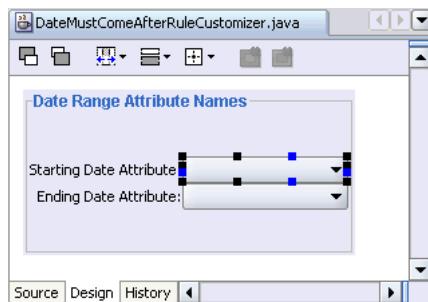
For easier reuse of your custom validation rules, you would typically package them into a JAR file for reference by applications that make use of the rules.

34.10.2 Adding a Design Time Bean Customizer for Your Rule

Since a validation rule class is a bean, you can implement a standard JavaBean customizer class to improve the design time experience of setting the bean properties. In the example of the `DateMustComeAfter` rule in the previous section, the two properties that the developers must configure are the `initialDateAttrName` and `laterDateAttrName` properties.

[Figure 34-3](#) illustrates using JDeveloper's visual designer for Swing to create a `DateMustComeAfterRuleCustomizer` using a `JPanel` with a titled border containing two `JLabel` prompts and two `JComboBox` controls for the dropdown lists. The code in the class populates the dropdown lists with the names of the Date-valued attributes of the current entity object being edited in the IDE. This will allow a developer who adds a `DateMustComeAfterRule` validation to their entity object to easily pick which date attributes should be used for the starting and ending dates for validation.

Figure 34–3 Using JDeveloper's Swing Visual Designer to Create Validation Rule Customizer



To associate a customizer with your `DateMustComeAfterRule` Java Bean, you follow the standard practice of creating a `BeanInfo` class. As shown in [Example 34–23](#), the `DateMustComeAfterRuleBeanInfo` returns a `BeanDescriptor` that associates the customizer class with the `DateMustComeAfter` bean class.

You would typically package your customizer class and this bean info in a separate JAR file for design-time-only use.

Example 34–23 BeanInfo to Associate a Customizer with a Custom Validation Rule

```
package oracle.fodemo...frameworkExt.rules;
import java.beans.BeanDescriptor;
import java.beans.SimpleBeanInfo;
public class DateMustComeAfterRuleBeanInfo extends SimpleBeanInfo {
    public BeanDescriptor getBeanDescriptor() {
        return new BeanDescriptor(DateMustComeAfterRule.class,
                               DateMustComeAfterRuleCustomizer.class);
    }
}
```

34.10.3 Registering and Using a Custom Rule in JDeveloper

After you've created a custom validation rule, you can add it to the project or application level in the JDeveloper IDE so that other developers can use the rule declaratively.

To register a custom validation rule in a project containing entity objects:

1. In the Application Navigator, right-click the desired project, and choose **Project Properties** from the context menu.
2. In the Project Properties dialog, expand **Business Components**, and select **Registered Rules**.
3. On the Registered Rules page, click **Add**.
4. In the Register Validation Rule dialog, browse to find the validation rule you created as in [Section 34.10.1, "How To Create a Custom Validation Rule"](#) and click **OK**.

To register a custom validator at the IDE level:

1. From the **Tools** menu, choose **Preferences**.
2. From the **Business Components > Register Rules** page, you can add a one or more validation rules.

When adding a validation rule, provide the fully-qualified name of the validation rule class, and supply a validation rule name that will appear in JDeveloper's list of available validators.

34.11 Creating New History Types

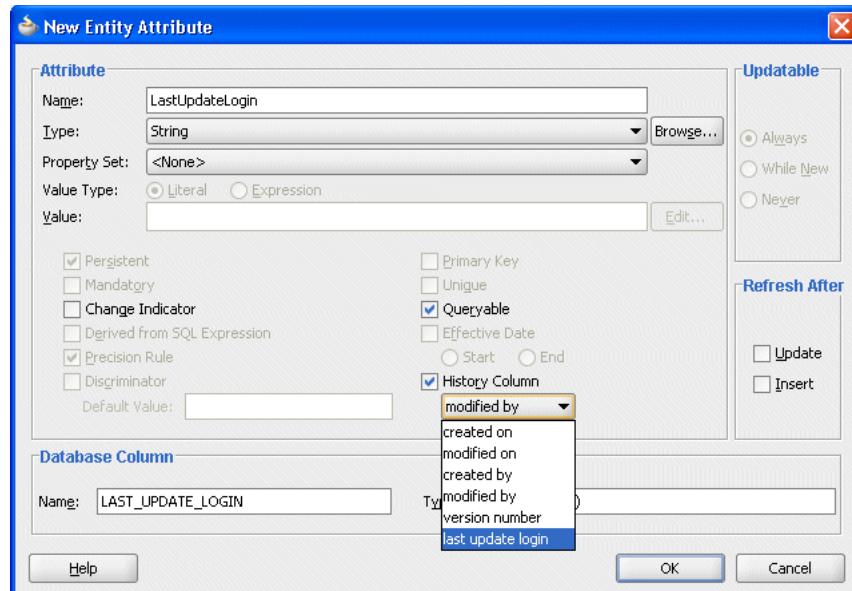
History types are used to track data specific to a point in time. JDeveloper ships with a number of history types, but you can also create your own. For more information on the standard history types and how to use them, see [Section 4.10.12, "How to Track Created and Modified Dates Using the History Column"](#).

34.11.1 How to Create New History Types

You are not limited to the history types provided, you can add or remove custom history types using the History Types page in the Preferences dialog, and then write custom Java code to implement the desired behavior. The code to handle custom history types should be written in your application-wide entity base class for reuse.

[Figure 34–5](#) shows a custom type called `last_update_login` with type Id of 11. Assume that `last_update_login` is a foreign key in the `FND_LOGINS` table.

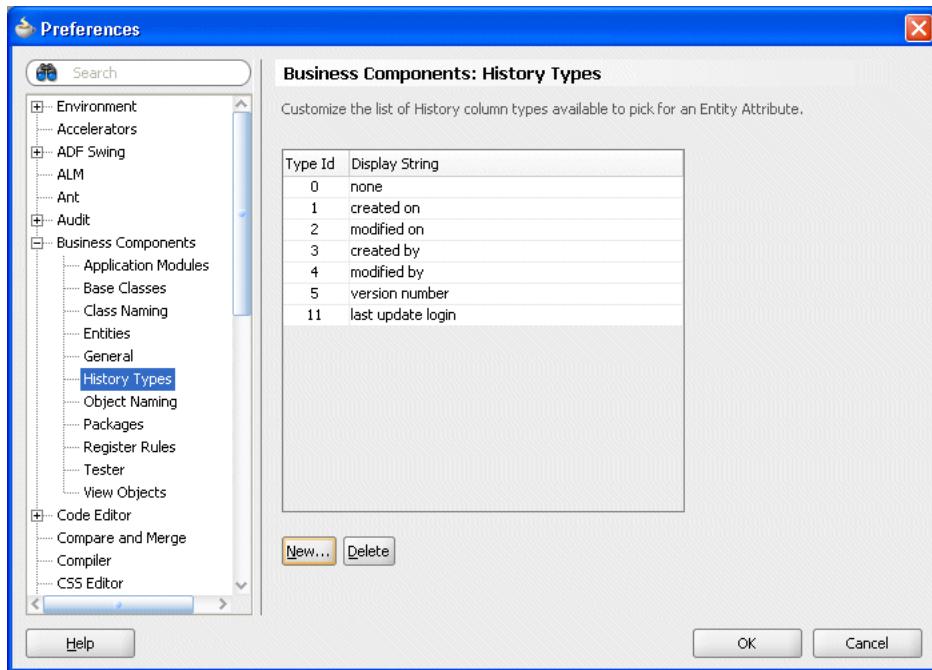
Figure 34–4 *New history types appear in the editor.*



To create a custom history type:

1. From the Tools menu, choose **Preferences**.
2. In the Preferences dialog, expand **Business Components**, and click **History Types**.
3. On the History Types page, click **New**.
4. In the Create History Type dialog, enter a string value for the name (spaces are allowed) and a numerical Id.

The **Type Id** must be an integer between 11 and 126. The numerical values 0-10 are reserved for internal use. The display string is displayed in the **History Column** dropdown list the next time you use the Edit Attribute dialog.

Figure 34–5 Creating new history types

5. Open the EntityImpl.java file and add a definition similar to the one in [Example 34–24](#).

Example 34–24 History type definition

```
private static final byte LASTUPDATELOGIN_HISTORY_TYPE = 11;
```

6. Override the getHistoryContextForAttribute(AttributeDefImpl attr) method in the EntityImpl base class with code similar to [Example 34–25](#).

Example 34–25 Overriding getHistoryContextForAttribute()

```
@Override
protected Object getHistoryContextForAttribute(AttributeDefImpl attr) {
    if (attr.getHistoryKind() == LASTUPDATELOGIN_HISTORY_TYPE) {
        // Custom History type logic goes here
    }
    else {
        return super.getHistoryContextForAttribute(attr);
    }
}
```

34.11.2 How to Remove a History Type

Because they are typically used for auditing values over the life of an application, it is rare that you would want to remove a history type. However, in the event that you need to do so, perform the following tasks:

1. Remove the history type from the JDeveloper history types list in the Preferences dialog.
2. Remove any custom code you implemented to support the history type in the base EntityImpl.getHistoryContextForAttribute method.

3. Remove all usages of the history type in the entity attribute metadata. Any attribute that you have defined to use this history type must be edited.

To remove a history type from the JDeveloper history types list:

1. From the **Tools** menu, choose **Preferences**.
2. In the Preferences dialog, expand **Business Components**, and click **History Types**.
3. On the History Types page, select the history type that you want to remove and click **Delete**.

Advanced View Object Techniques

This chapter describes advanced techniques you can use while designing and working with your view objects.

This chapter includes the following sections:

- [Section 35.1, "Advanced View Object Concepts and Features"](#)
- [Section 35.2, "Tuning Your View Objects for Best Performance"](#)
- [Section 35.3, "Using Expert Mode for Full Control Over SQL Query"](#)
- [Section 35.4, "Generating Custom Java Classes for a View Object"](#)
- [Section 35.5, "Working Programmatically with Multiple Named View Criteria"](#)
- [Section 35.6, "Performing In-Memory Sorting and Filtering of Row Sets"](#)
- [Section 35.7, "Using View Objects to Work with Multiple Row Types"](#)
- [Section 35.8, "Reading and Writing XML"](#)
- [Section 35.9, "Using Programmatic View Objects for Alternative Data Sources"](#)
- [Section 35.10, "Creating a View Object with Multiple Updatable Entities"](#)
- [Section 35.11, "Declaratively Preventing Insert, Update, and Delete"](#)

35.1 Advanced View Object Concepts and Features

This section describes a number of interesting view object concepts and features that have not been discussed in previous chapters.

35.1.1 Using a Max Fetch Size to Only Fetch the First N Rows

The default maximum fetch size of a view object is minus one (-1), which indicates there should be no limit to the number of rows that can be fetched. Keep in mind that by default, rows are fetched as needed, so -1 does not imply a view object will necessarily fetch all the rows. It simply means that if you attempt to iterate through all the rows in the query result, you will get them all.

However, you might want to put an upper bound on the maximum number of rows that a view object will retrieve. If you write a query containing an ORDER BY clause and only want to return the first N rows to display the "Top-N" entries in a page, you can use the overview editor for the view object to specify a value for the **Only up to row number** field in the Tuning section of the General page. For example, to fetch only the first five rows, you would enter "5" in this field. This is equivalent to calling the `setMaxFetchSize()` method on your view object to set the maximum fetch size to 5. The view object will stop fetching rows when it hits the maximum fetch size. Often

you will combine this technique with specifying a **Query Optimizer Hint** of FIRST_ROWS also on the Tuning section of the General page of the overview editor. This gives a hint to the database that you want to retrieve the first rows as quickly as possible, rather than trying to optimize the retrieval of all rows.

35.1.2 Consistently Displaying New Rows in View Objects Based on the Same Entity

When multiple instances of entity-based view objects in an application module are based on the same underlying entity object, a new row created in one of them can be automatically added (without having to re-query) to the row sets of the others to keep your user interface consistent or simply to consistently reflect new rows in different application pages for a pending transaction. Consider the Fusion Order Demo application's `orderSummary.jspx` page that displays a customer's list of orders. If the customer goes to create a new order, this task is performed through a different view object and handled by a custom application module method. Using this view object new row consistency feature, the newly created order automatically appears in the customer's list of open orders on the `orderSummary.jspx` page without having to re-query the database.

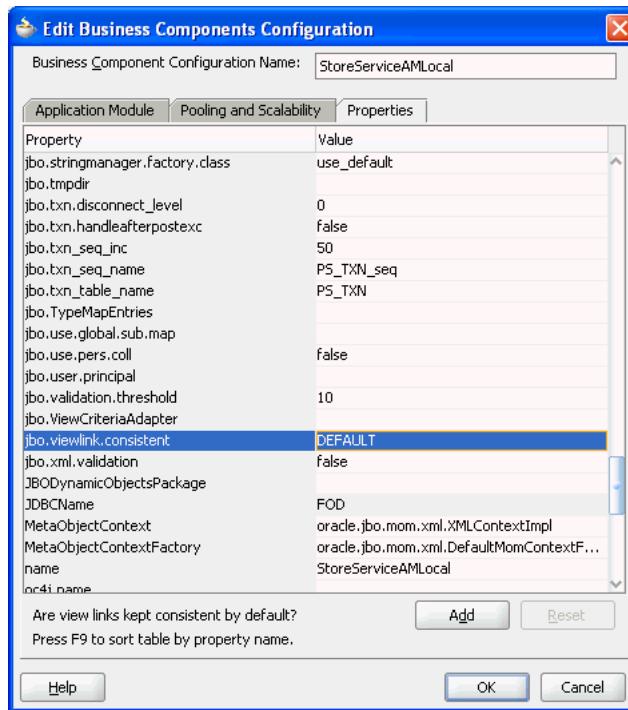
For historical reasons, this capability is known as the *view link consistency* feature because in prior releases of Oracle ADF the addition of new rows to other relevant row sets only was supported for detail view object instances in a view link based on an association. Now this view link consistency feature works for any view objects for which it is enabled, regardless of whether they are involved in a view link or not.

35.1.2.1 How View Link Consistency Mode Works

Consider two entity-based view objects `OrdersViewSummary` and `OrdersView` both based on the same underlying `Orders` entity object. When a new row is created in a row set for one of these view objects (like `OrdersView`) and the row's primary key is set, any of the *other* row sets for view objects based on the same `Orders` entity object (like `OrdersViewSummary`) receive an event indicating a new row has been created. If their view link consistency flag is enabled, then a copy of the new row is inserted into their row set as well.

35.1.2.2 Understanding the Default View Link Consistency Setting and How to Change It

You can use the Edit Business Components Configuration dialog to control the default setting for the view link consistency feature using the `jbo.viewlink.consistent` configuration parameter, as shown in [Figure 35-1](#).

Figure 35–1 jbo.viewlink.consistent Property Setting in Configuration Editor

To display the configuration editor, right-click the application module in the Application Navigator and choose **Configurations**. Then, in the Manage Configurations dialog, select the configuration and click **Edit**. In the Edit Business Components Configuration dialog, select the Properties tab. The default setting for this parameter is the word "DEFAULT" which has the following meaning. If your view object has:

- A single entity usage, view link consistency is enabled
- Multiple entity usages, and:
 - If all secondary entity usages are marked as contributing reference information, then view link consistency is enabled
 - If any secondary entity usage marked as **not** being a reference view link consistency is disabled.

You can globally disable this feature by setting the `jbo.viewlink.consistent` to the value `false` in your configuration. Conversely, you could globally enable this feature by setting `jbo.viewlink.consistent` to the value `true`, but Oracle does **not** recommend doing this. Doing so would force view link consistency to be set on for view objects with secondary entity usages that are not marked as a reference which presently do not support the view link consistency feature well.

To set the feature programmatically, use the `setAssociationConsistent()` API on any RowSet. When you call this method on a view object, it affects its default row set.

35.1.2.3 Using a RowMatch to Qualify Which New, Unposted Rows Get Added to a Row Set

If a view object has view link consistency enabled, any new row created by another view object based on the same entity object is added to its row set. By default the mechanism adds new rows in an unqualified way. If your view object has a

design-time WHERE clause that queries only a certain subset of rows, you can apply a RowMatch object to your view object to perform the same filtering in-memory. The filtering expression of the RowMatch object you specify prevents new rows from being added that wouldn't make sense to appear in that view object.

For example, an OrdersByStatus view object might include a design time WHERE clause like this:

```
WHERE /* ... */ AND STATUS LIKE NVL(:StatusCode, '%')
```

Its custom Java class overrides the `create()` method as shown in [Example 35–1](#) to force view link consistency to be enabled. It also applies a RowMatch object whose filtering expression matches rows whose `Status` attribute matches the value of the `:StatusCode` named bind variable (or matches any row if `:StatusCode = '%'`). This RowMatch filter is used by the view link consistency mechanism to qualify the row that is a candidate to add to the row set. If the row qualifies by the RowMatch, it is added. Otherwise, it is not.

Example 35–1 Providing a Custom RowMatch to Control Which New Rows are Added

```
// In OrdersByStatusImpl.java
protected void create() {
    super.create();
    setAssociationConsistent(true);
    setRowMatch(new RowMatch("Status = :StatusCode or :StatusCode = '%'"));
}
```

See [Section 35.6.4, "Performing In-Memory Filtering with RowMatch"](#) for more information on creating and using a RowMatch object. For a list of supported SQL operators see [Table 35–2](#). For a list of supported SQL function, see [Table 35–3](#).

Note: If the RowMatch facility does not provide enough control, you can override the view object's `rowQualifies()` method to implement a custom filtering solution. Your code can determine whether a new row qualifies to be added by the view link consistency mechanism or not.

35.1.2.4 Setting a Dynamic Where Clause Disables View Link Consistency

If you call `setWhereClause()` on a view object to set a dynamic where clause, the view link consistency feature is disabled on that view object. If you have provided an appropriate custom RowMatch object to qualify new rows for adding to the row set, you can call `setAssociationConsistent(true)` after `setWhereClause()` to re-enable view link consistency.

35.1.2.5 New Row from Other View Objects Added at the Bottom

If a row set has view link consistency enabled, then new rows added due to creation by other row sets are added to the bottom of the row set.

35.1.2.6 New, Unposted Rows Added to Top of RowSet when Re-Executed

If a row set has view link consistency enabled, then when you call the `executeQuery()` method, any qualifying new, unposted rows are added to the top of the row set before the queried rows from the database are added.

35.1.3 Understanding View Link Accessors Versus Data Model View Link Instances

View objects support two different styles of master-detail coordination:

- View link *instances* for data model master/detail coordination, as described in [Section 35.1.3.1, "Enabling a Dynamic Detail Row Set with Active Master/Detail Coordination"](#).
- View link *accessor* attributes for programmatically accessing detail row sets on demand, as described in [Section 35.1.3.2, "Accessing a Stable Detail Row Set Using View Link Accessor Attributes"](#).
- You can combine both styles, as described in [Section 35.1.3.3, "Accessor Attributes Create Distinct Row Sets Based on an Internal View Object"](#).

35.1.3.1 Enabling a Dynamic Detail Row Set with Active Master/Detail Coordination

When you add a view link instance to your application module's data model, you connect two specific view object instances and indicate that you want active master/detail coordination between the two. At runtime the view link instance in the data model facilitates the eventing that enables this coordination. Whenever the current row is changed on the master view object instance, an event causes the detail view object to be refreshed by automatically invoking `executeQuery()` with a new set of bind parameters for the new current row in the master view object.

A key feature of this data model master/detail is that the master and detail view object instances are stable objects to which client user interfaces can establish bindings. When the current row changes in the master — instead of producing a *new* detail view object instance — the existing detail view object instance updates its default row set to contain the set of rows related to the new current master row. In addition, the user interface binding objects receive events that allow the display to update to show the detail view object's refreshed row set.

Another key feature that is exclusive to data model master/detail is that a detail view object instance can have multiple master view object instances. For example, an `PaymentOptions` view object instance may be a detail of *both* a `Customers` and a `Orders` view object instance. Whenever the current row in *either* the `Customers` or `Orders` view object instance changes, the default row set of the detail `PaymentOptions` view object instance is refreshed to include the row of expertise area information for the current technician and the current product. See [Section 35.1.6, "Setting Up a Data Model with Multiple Masters"](#) for details on setting up a detail view object instance with multiple-masters.

35.1.3.2 Accessing a Stable Detail Row Set Using View Link Accessor Attributes

When you need to programmatically access the detail row set related to a view object row by virtue of a view link, you can use the view link accessor attribute. You specify the finder name of the view link accessor attribute from the overview editor for the view link. Click the **Edit** icon in the Accessors section of the Relationship page and in the Edit View Link Properties dialog, edit the name of the view link accessor attribute.

[Example 35–2](#) shows the XML for the view link that defines the `_findername` value of the `<Attr>` element.

Example 35–2 View Link Accessor Attribute Name

```
<ViewLinkDefEnd
    Name="Orders"
    Cardinality="1"
    Owner="devguide.advanced.multiplemasters.Orders"
```

```
Source="true">
<AttrArray Name="Attributes">
  <Item Value="devguide.advanced.multiplemasters.Orders.PaymentOptionId"/>
</AttrArray>
<DesignTime>
  <Attr Name="_minCardinality" Value="1"/>
  <Attr Name="_isUpdateable" Value="true"/>
  <Attr Name="_finderName" Value="Orders"/>
</DesignTime>
</ViewLinkDefEnd>
```

Assuming you've named your accessor attribute *AccessorAttrName*, you can access the detail row set using the generic `getAttribute()` API like:

```
RowSet detail = (RowSet)currentRow.getAttribute("AccessorAttrName");
```

If you've generated a custom view row class for the master view object and exposed the getter method for the view link accessor attribute on the client view row interface, you can write strongly-typed code to access the detail row set like this:

```
RowSet detail = (RowSet)currentRow.getAccessorAttrName();
```

Unlike the data model master/detail, programmatic access of view link accessor attributes does not require a detail view object instance in the application module's data model. Each time you invoke the view link accessor attribute, it returns a RowSet containing the set of detail rows related to the master row on which you invoke it.

Using the view link accessor attribute, the detail data rows are stable. As long as the attribute value(s) involved in the view link definition in the master row remain unchanged, the detail data rows will not change. Changing of the current row in the master does not affect the detail row set which is "attached" to a given master row. For this reason, in addition to being useful for general programmatic access of detail rows, view link accessor attributes are appropriate for UI objects like the tree control, where data for each master node in a tree needs to retain its distinct set of detail rows.

35.1.3.3 Accessor Attributes Create Distinct Row Sets Based on an Internal View Object

When you *combine* the use of data model master/detail with programmatic access of detail row sets using view link accessor, it is even more important to understand that they are distinct mechanisms. For example, imagine that you:

- Define `PersonsVO` and `OrdersVO` view objects
- Define a view link between them, naming the view link accessor `PersonsToOrders`
- Add instances of them to an application module's data model named `master` (of type `PersonsVO`) and `detail` (of type `OrdersVO`) coordinated actively by a view link instance.

If you find a person in the master view object instance, the detail view object instance updates as expected to show the corresponding orders. At this point, if you invoke a custom method that programmatically accesses the `PersonsToOrders` view link accessor attribute of the current `PersonsVO` row, you get a RowSet containing the set of `OrdersVO` rows. You might reasonably expect this programmatically accessed RowSet to have come from the detail view object instance in the data model, but this is not the case.

The RowSet returned by a view link accessor always originates from an *internally created* view object instance, not one you added to the data model. This internal

view object instance is created as needed and added with a system-defined name to the root application module.

The principal reason a distinct, internally-created view object instance is used is to guarantee that it remains unaffected by developer-related changes to their own view objects instances in the data model. For example, if the view row were to use the detail view object in the data model for view link accessor RowSet, the resulting row set could be inadvertently affected when the developer dynamically:

1. Adds a WHERE clause with new named bind parameters

If such a view object instance were used for the view link accessor result, unexpected results or an error could ensue because the dynamically-added WHERE clause bind parameter values have not been supplied for the view link accessor's RowSet: they were only supplied for the default row set of the detail view object instance in the data model.

2. Adds an additional master view object instance for the detail view object instance in the data model.

In this scenario, the semantics of the accessor would be changed. Instead of the accessor returning OrdersVO rows for the current PersonsVO row, it could all of a sudden start returning *only* the OrdersVO rows for the current PersonsVO that were created by a current logged in customer, for example.

3. Removes the detail view object instance or its containing application module instance.

In this scenario, all rows in the programmatically-accessed detail RowSet would become invalid.

Furthermore, Oracle ADF needs to distinguish between the data model master/detail and view link accessor row sets for certain operations. For example, when you create a new row in a detail view object, the framework automatically populates the attributes involved in the view link with corresponding values of the master. In the data model master/detail case, it gets these values from the current row(s) of the possibly multiple master view object instances in the data model. In the case of creating a new row in a RowSet returned by a view link accessor, it populates these values from the master row on which the accessor was called.

35.1.4 Presenting and Scrolling Data a Page at a Time Using the Range

To present and scroll through data a page at a time, you can configure a view object to manage for you an appropriately-sized range of rows. The range facility allows a client to easily display and update a subset of the rows in a row set, as well as easily scroll to subsequent pages N rows at a time. You call `setRangeSize()` to define how many rows of data should appear on each page. The default range size is one (1) row. A range size of minus one (-1) indicates the range should include all rows in the row set.

Note: When using the ADF Model layer's declarative data binding, the iterator binding in the page definition has a RangeSize property. At runtime, the iterator binding invokes the `setRangeSize()` method on its corresponding row set iterator, passing the value of this RangeSize property. The ADF design time by default sets this RangeSize property to 10 rows for most iterator bindings. An exception is the range size specified for a List binding to supply the set of valid values for a UI component like a dropdown list. In this case, the default range size is minus one (-1) to allow the range to include all rows in the row set.

When you set a range size greater than one, you control the row set paging behavior using the iterator mode. The two iterator mode flags you can pass to the `setIterMode()` method are:

- `RowIterator.ITER_MODE_LAST_PAGE_PARTIAL`

In this mode, the last page of rows may contain fewer rows than the range size. For example, if you set the range size to 10 and your row set contains 23 rows, the third page of rows will contain only three rows. This is the style that works best for Fusion web applications.

- `RowIterator.ITER_MODE_LAST_PAGE_FULL`

In this mode, the last page of rows is kept full, possibly including rows at the top of the page that had appeared at the bottom of the previous page. For example, if you set the range size to 10 and your row set contains 23 rows, the third page of rows will contain 10 rows, the first seven of which appeared as the last seven rows of page two. This is the style that works best for desktop-fidelity applications using Swing.

35.1.5 Efficiently Scrolling Through Large Result Sets Using Range Paging

As a general rule, for highest performance, Oracle recommends building your application in a way that avoids giving the end user the opportunity to scroll through very large query results. To enforce this recommendation, call the `getEstimatedRowCount()` method on a view object to determine how many rows would be returned by the user's query *before* actually executing the query and allowing the user to proceed. If the estimated row count is unreasonably large, your application can demand that the end-user provide additional search criteria.

However, when you *must* work with very large result sets, typically over 100 rows, you can use the view object's access mode called "range paging" to improve performance. The feature allows your applications to page back and forth through data, a range of rows at a time, in a way that is more efficient for large data sets than the default "scrollable" access mode.

The range paging access mode is typically used for paging through read-only row sets, and often is used with read-only view objects. You allow the user to find the row they are looking for by paging through a large row set with range paging access mode, then you use the Key of that row to find the selected row in a different view object for editing.

Range paging for view objects supports a standard access mode and a variation of the standard access mode that combines the benefits of range paging and result set scrolling with a minimum number of visits to the database. These modes for the view object range paging feature include:

- RANGE_PAGING, standard access mode fetches the number of rows specified by a range size. In this mode, the number of rows that may be scrolled without requerying the database is determined by a range size that you set. The default is to fetch a single row, but it is expected that you will set a range size equal to the number of rows you want to be able to display to the user before they scroll to the next result set. The application requeries the database each time a row outside of the range is accessed by the end user. Thus, scrolling backward and forward through the row set will requery the database. For clarification about this database-centric paging strategy, see [Section 35.1.5.1, "Understanding How to Oracle Supports "TOP-N" Queries"](#).
- RANGE_PAGING_INCR, incremental access mode gives the UI designer more flexibility for the number of rows to display at a time while keeping database queries to a minimum. In this mode, the UI incrementally displays the result set from the memory cache and thus supports scrolling within a single database query. The number of rows that the end user can scroll though in a single query is determined by the range size and a range paging cache factor that you set. For example, suppose that you set the range size to 4 and the cache factor to 5. Then, the maximum number of rows to cache in memory will be $4 \times 5 = 20$. For further explanation of the caching behavior, see [Section 35.1.5.4, "What Happens When View Rows are Cached When Using Range Paging"](#).

Caution: Additionally, the view object supports a RANGE_PAGING_AUTO_POST access mode to accommodate the inserting and deleting of rows from the row set. This mode behaves like the RANGE_PAGING mode, except that it eagerly calls `postChanges()` on the database transaction whenever any changes are made to the row set. However, this mode is typically not appropriate for use in Fusion web applications unless you can guarantee that the transaction will definitely be committed or rolled-back during the same HTTP request. Failure to heed this advice can lead to strange results in an environment where both application modules and database connections can be pooled and shared serially by multiple different clients.

35.1.5.1 Understanding How to Oracle Supports "TOP-N" Queries

The Oracle database supports a feature called a "Top-N" query to efficiently return the first N ordered rows in a query. For example, if you have a query like:

```
SELECT EMPNO, ENAME, SAL FROM EMP ORDER BY SAL DESC
```

If you want to retrieve the top 5 employees by salary, you can write a query like:

```
SELECT * FROM (
  SELECT X.* , ROWNUM AS RN FROM (
    SELECT EMPNO, ENAME, SAL FROM EMP ORDER BY SAL DESC
  ) X
) WHERE RN <= 5
```

which gives you results like:

EMPNO	ENAME	SAL	RN
7839	KING	5000	1
7788	SCOTT	3000	2
7902	FORD	3000	3
7566	JONES	2975	4

The feature is not only limited to retrieving the first N rows in order. By adjusting the criteria in the outermost WHERE clause you can efficiently retrieve any range of rows in the query's sorted order. For example, to retrieve rows 6 through 10 you could alter the query this way:

```
SELECT * FROM (
  SELECT X.* , ROWNUM AS RN FROM (
    SELECT EMPNO, ENAME, SAL FROM EMP ORDER BY SAL DESC
  ) X
) WHERE RN BETWEEN 6 AND 10
```

Generalizing this idea, if you want to see page number P of the query results, where each page contains R rows, then you would write a query like:

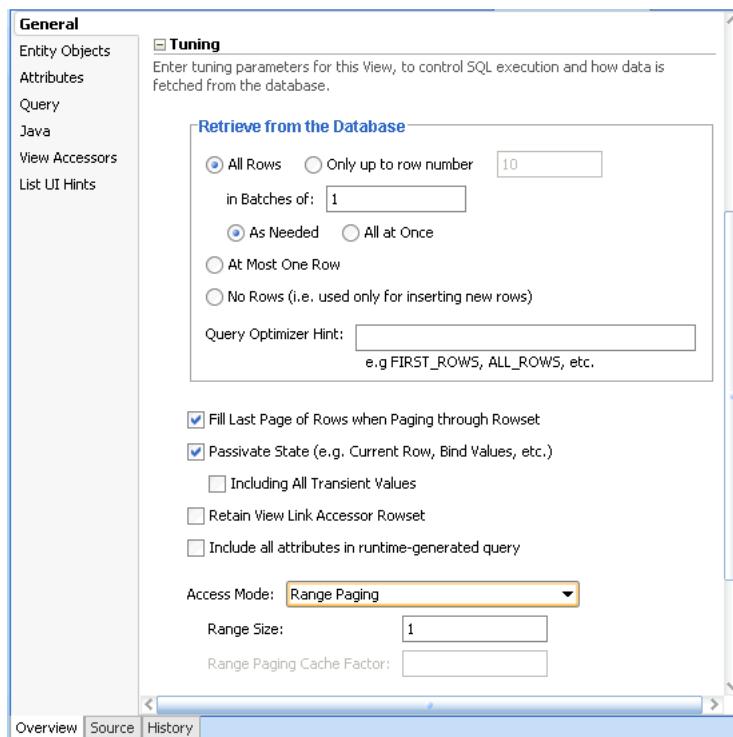
```
SELECT * FROM (
  SELECT X.* , ROWNUM AS RN FROM (
    SELECT EMPNO, ENAME, SAL FROM EMP ORDER BY SAL DESC
  ) X
) WHERE RN BETWEEN ((:P - 1) * :R) + 1 AND (:P) * :R
```

As the result set you consider grows larger and larger, it becomes more and more efficient to use this technique to page through the rows. Rather than retrieving hundreds or thousands of rows over the network from the database, only to display ten of them on the page, instead you can produce a clever query to retrieve only the R rows on page number P from the database. No more than a handful of rows at a time needs to be returned over the network when you adopt this strategy.

To implement this database-centric paging strategy in your application, you could handcraft the clever query yourself and write code to manage the appropriate values of the $:R$ and $:P$ bind variables. Alternatively, you can use the view object's range paging access mode, which implements it automatically for you.

35.1.5.2 How to Enable Range Paging for a View Object

You can use the Tuning panel of the overview editor for the view object to set the access mode to either standard range paging or incremental range paging. The **Range Paging Cache Factor** field is only editable when you select **Range Paging Incremental**. Figure 35–2 shows the view object's **Access Mode** set to **Range Paging** (standard mode) with the default range size of 1. To understand the row set caching behavior of both access modes, see [Section 35.1.5.4, "What Happens When View Rows are Cached When Using Range Paging"](#).

Figure 35–2 Access Mode in the Overview Editor for the View Object

To programmatically enable standard range paging for your view object, first call `setRangeSize()` to define the number of rows per page, then call the following method with the desired mode:

```
yourViewObject.setAccessMode(RowSet.RANGE_PAGING | RANGE_PAGING_INCR);
```

If you set `RANGE_PAGING_INCR`, then you must also call the following method to set the cache factor for your defined range size:

```
yourViewObject.setRangePagingCacheFactor(int f);
```

35.1.5.3 What Happens When You Enable Range Paging

When a view object's access mode is set to `RANGE_PAGING`, the view object takes its default query like:

```
SELECT EMPNO, ENAME, SAL FROM EMP ORDER BY SAL DESC
```

and automatically "wraps" it to produce a Top-N query.

For best performance, the statement uses a combination of greater than and less than conditions instead of the `BETWEEN` operator, but the logical outcome is the same as the Top-N wrapping query you saw above. The actual query produced to wrap a base query of:

```
SELECT EMPNO, ENAME, SAL FROM EMP ORDER BY SAL DESC
```

looks like this:

```
SELECT * FROM (
  SELECT /*+ FIRST_ROWS */ IQ.*, ROWNUM AS Z_R_N FROM (
    SELECT EMPNO, ENAME, SAL FROM EMP ORDER BY SAL DESC
  ) IQ WHERE ROWNUM < :0)
WHERE Z_R_N > :1
```

The two bind variables are bound as follows:

- `:1` index of the first row in the current page
- `:0` is bound to the last row in the current page

35.1.5.4 What Happens When View Rows are Cached When Using Range Paging

When a view object operates in `RANGE_PAGING` access mode, it only keeps the current range (or "page") of rows in memory in the view row cache at a time. That is, if you are paging through results ten at a time, then on the first page, you'll have rows 1 through 10 in the view row cache. When you navigate to page two, you'll have rows 11 through 20 in the cache. This also can help make sure for large row sets that you don't end up with tons of rows cached just because you want to preserve the ability to scroll backwards and forwards.

When a view object operates in `RANGE_PAGING_INCR` access mode, the cache factor determines the number of rows to cache in memory for a specific range size. For example, suppose the range size is set to 4 and cache factor to 5. Then, the memory will keep at most $4*5 = 20$ rows in its collection. In this example, when the range is refreshed for the first time, the memory will have just four rows even though the range paging query is bound to retrieve rows 0 to 19 (for a total of twenty rows). When the range is scrolled past the forth row, more rows will be read in from the current result set. This will continue until all twenty rows from the query result are read. If the user's action causes the next set of rows to be retrieved, the query will be re-executed with the new row number bind values. The exact row number bind values are determined by the new range-start and the number of rows that can be retained from the cache. For example, suppose all twenty rows have been filled up and the user asks to move the range-start to 18 (0-based). This means that memory can retain row 18 and row 19 and will need two more rows to fill the range. The query is re-executed for rows 20 and 21.

35.1.5.5 How to Scroll to a Given Page Number Using Range Paging

When a view object operates in `RANGE_PAGING` access mode, to scroll to page number N call its `scrollToRangePage()` method, passing N as the parameter value.

35.1.5.6 Estimating the Number of Pages in the Row Set Using Range Paging

When a view object operates in `RANGE_PAGING` access mode, you can access an estimate of the total number of pages the entire query result would produce using the `getEstimatedRangePageCount()` method.

35.1.5.7 Understanding the Tradeoffs of Using a Range Paging Mode

You might ask yourself, "Why wouldn't I *always* want to use `RANGE_PAGING` or `RANGE_PAGING_INCR` mode?" The answer is that using range paging potentially causes more overall queries to be executed as you are navigating forward and backward through your view object rows. You would want to avoid using `RANGE_PAGING` mode in these situations:

- You plan to read all the rows in the row set immediately (for example, to populate a dropdown list).

In this case your range size would be set to `-1` and there really is only a single "page" of all rows, so range paging does not add value.

- You need to page back and forth through a small-sized row set.

If you have 100 rows or fewer, and are paging through them 10 at a time, with `RANGE_PAGING` mode you will execute a query each time you go forward and

backward to a new page. Otherwise, in the default scrollable mode, you will cache the view object rows as you read them in, and paging backwards through the previous pages will not re-execute queries to show those already-seen rows. Alternatively, you can use RANGE_PAGING_INCR mode to allow scrolling through in-memory results based on a row set cache factor that you determine.

In the case of a very large (or unpredictably large) row set, the trade off of potentially doing a few more queries — each of which only returns up to the RangeSize number of rows from the database — is more efficient than trying to cache all of the previously-viewed rows. This is especially true if you allow the user to jump to an arbitrary page in the list of results. Doing so in default, scrollable mode requires fetching and caching all of the rows between the current page and the page the users jumps to. In RANGE_PAGING mode, it will ask the database just for the rows on that page. Then, if the user jumps back to a page of rows that they have already visited, in RANGE_PAGING mode, those rows get re-queried again since only the current page of rows is held in memory in this mode. The incremental range paging access mode RANGE_PAGING_INCR combines aspects of both standard range paging and scrollable access mode since it allows the application to cache more rows in memory and permits the user to jump to any combination of those rows without needing to requery.

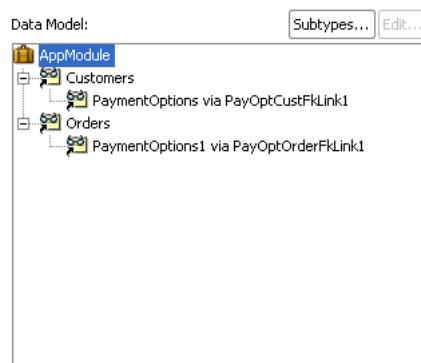
35.1.6 Setting Up a Data Model with Multiple Masters

When useful, you can set up your data model to have multiple master view object instances for the same detail view object instance. Consider view objects named Customers, Orders, and PaymentOptions with view links defined between:

- Customers and PaymentOptions
- Orders and PaymentOptions

[Figure 35–3](#) shows what the data model panel looks like when you've configured both Customers and Orders view object instances to be masters of the same PaymentOptions view object instance.

Figure 35–3 Multiple Master View Object Instances for the Same Detail



To set up the data model as shown in [Figure 35–3](#) open the overview editor for the application module and follow these steps in the Data Model Components section of the Data Model page:

1. Add an instance of the Customers view object to the data model.

Assume you name it Customers.

2. Add an instance of the Orders view object to the data model

Assume you name it Orders.

3. Select the Customers view object instance in the **Data Model** list
4. In the **Available View Objects** list, select the PaymentOptions view object indented beneath the Customers view object and enter the view object instance name of PaymentOptions in the **New Instance Name** field. Click **>** to shuttle it into data model as a detail of the existing Customers view object instance.
5. Select the Orders view object instance in the **Data Model** list
6. In the **Available View Objects** list, select the PaymentOptions view object indented beneath the Orders view object and enter the view object instance name of PaymentOptions in the **New Instance Name** field. Click **>** to shuttle it into data model as a detail of the existing Orders view object instance.

An alert will appear: **An instance of a View Object with the name PaymentOptions has already been used in the data model. Would you like to use the same instance?**
7. Click **Yes** to confirm you want the PaymentOptions view object instance to *also* be the detail of the Orders view object instance.

35.1.7 Understanding When You Can Use Partial Keys with `findByPrimaryKey()`

View objects based on multiple entity usages support the ability to find view rows by specifying a partially populated key. A partial key is a multi-attribute Key object with some of its attributes set to null. However, there are strict rules about what kinds of partial keys can be used to perform the `findByPrimaryKey()`.

If a view object is based on N entity usages, where $N > 1$, then the view row key is by default comprised of all of the primary key attributes from *all* of the participating entity usages. Only the ones from the *first* entity object are *required* to participate in the view row key, but by default all of them do.

If you allow the key attributes from some *secondary* entity usages to remain as key attributes at the view row level, then you should leave *all* of the attributes that form the primary key for that entity object as part of the view row key. Assuming you have left the one or more key attributes in the view row for M of the N entity usages, where ($M \leq N$), then you can use `findByPrimaryKey()` to find rows based on any subset of these M entity usages. Each entity usage for which you provide values in the Key object, requires that you must provide non-null values for *all* of the attributes in that entity's primary key.

You have to follow this rule because when a view object is based on at least one or more entity usages, its `findByPrimaryKey()` method finds rows by delegating to the `findByPrimaryKey()` method on the entity definition corresponding to the first entity usage whose attributes in the view row key are non-null. The entity definition's `findByPrimaryKey()` method requires all key attributes for any given entity object to be non-null in order to find the entity row in the cache.

As a concrete example, imagine that you have a `OrderInfoVO` view object with a `OrderEO` entity object as its primary entity usage, and an `AddressEO` entity as secondary reference entity usage. Furthermore, assume that you leave the **Key Attribute** property of *both* of the following view row attributes set to `true`:

- `OrderId` — primary key for the `OrderEO` entity
- `AddressId` — primary key for the `AddressEO` entity

The view row key will therefore be the `(OrderId, AddressId)` combination. When you do a `findByPrimaryKey()`, you can provide a Key object that provides:

- A completely specified key for the underlying `OrderEO` entity

```
Key k = new Key(new Object[]{new Number(200), null});
```

- A completely specified key for the underlying AddressEO entity

```
Key k = new Key(new Object[]{null, new Number(118)});
```

- A completely specified key for both entities

```
Key k = new Key(new Object[]{new Number(200), new Number(118)});
```

When a valid partial key is specified, the `findByPrimaryKey()` method can return multiple rows as a result, treating the missing entity usage attributes in the Key object as a wildcard.

35.1.8 Creating Dynamic Attributes to Store UI State

You can add one or more dynamic attributes to a view object at runtime using the `addDynamicAttribute()` method. Dynamic attributes can hold any serializable object as their value. Typically, you will consider using dynamic attributes when writing generic framework extension code that requires storing some additional per-row transient state to implement a feature you want to add to the framework in a global, generic way.

35.1.9 Working with Multiple Row Sets and Row Set Iterators

While you typically work with a view object's *default* row set, you can call the `createRowSet()` method on the `ViewObject` interface to create secondary, named row sets based on the same view object's query. One situation where this could make sense is when your view object's SQL query contains named bind variables. Since each `RowSet` object stores its own copy of bind variable values, you could use a single view object to produce and process multiple row sets based on different combinations of bind variable values. You can find a named row set you've created using the `findRowSet()` method. When you're done using a secondary row set, call its `closeRowSet()` method.

For any `RowSet`, while you typically work with its *default* row set iterator, you can call the `createRowSetIterator()` method of the `RowSet` interface to create secondary, named row set iterators. You can find a named row set iterator you've created using the `findRowSetIterator()` method. When you're done using a secondary row set iterator, call its `closeRowSetIterator()` method.

Performance Tip: When you need to perform programmatic iteration over a result set, create a secondary iterator to avoid disturbing the current row of the *default* row set iterator. For example, through the ADF Model declarative data binding layer, user interface pages in your application work with the default row set iterator of the default row set of view objects in the application module's data model. In this scenario, if you did not create a secondary row set iterator for the business logic you write to iterate over a view object's default row set, you would consequently change the current row of the *default* row set iterator used by the user interface layer.

35.1.10 Optimizing View Link Accessor Access By Retaining the Row Set

Each time you retrieve a view link accessor row set, by default the view object creates a new `RowSet` object to allow you to work with the rows. This does *not* imply re-executing the query to produce the results each time, only creating a new instance of a `RowSet` object with its default iterator reset to the "slot" before the first row. To force

the row set to refresh its rows from the database, you can call its `executeQuery()` method.

You can enable caching of the view link accessor row set when you do not want the application to incur the small amount of overhead associated with creating new detail row sets. For example, because view accessor row sets remain stable as long as the master row view accessor attribute remains unchanged, it would not be necessary to recreate a new row set for UI components, like the tree control, where data for each master node in a tree needs to retain its distinct set of detail rows. The view link accessor's detail row set can also be accessed programmatically. In this case, if your application makes numerous calls to the same view link accessor attributes, you can consider caching the view link accessor row set. This style of managing master-detail coordination differs from creating view link instances in the data model, as explained in [Section 35.1.3, "Understanding View Link Accessors Versus Data Model View Link Instances"](#).

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

Alternatively, you can enable a custom Java class for your view object, override the `create()` method, and add a line after `super.create()` that calls the `setViewLinkAccessorRetained()` method passing `true` as the parameter. It affects all view link accessor attributes for that view object.

When this feature is enabled for a view object, since the view link accessor row set is not recreated each time, the current row of its default row set iterator is also retained as a side-effect. This means that your code will need to explicitly call the `reset()` method on the row set you retrieve from the view link accessor to reset the current row in its default row set iterator back to the "slot" before the first row.

Note, however, that with accessor retention enabled, your failure to call `reset()` each time before you iterate through the rows in the accessor row set can result in a subtle, hard-to-detect error in your application. For example, if you iterate over the rows in a view link accessor row set like this, for example to calculate some aggregate total:

```
RowSet rs = (RowSet)row.getAttribute("OrdersShippedToPurchaser");
while (rs.hasNext()) {
    Row r = rs.next();
    // Do something important with attributes in each row
}
```

The first time you work with the accessor row set the code will work. However, since the row set (and its default row set iterator) are retained, the second and subsequent times you access the row set the current row will already be at the end of the row set and the while loop will be skipped since `rs.hasNext()` will be `false`. Instead, with this feature enabled, write your accessor iteration code like this:

```
RowSet rs = (RowSet)row.getAttribute("OrdersShippedToPurchaser");
rs.reset(); // Reset default row set iterator to slot before first row!
while (rs.hasNext()) {
    Row r = rs.next();
    // Do something important with attributes in each row
}
```

Recall that if view link consistency is on, when the accessor is retained the new unposted rows will show up at the end of the row set. This is slightly different from when the accessor is not retained (the default), where new unposted rows will appear at the beginning of the accessor row set.

35.2 Tuning Your View Objects for Best Performance

You can use view objects to read rows of data, create and store rows of transient data, as well as automatically coordinate inserts, updates, and deletes made by end users with your underlying business objects. How you design and use your view objects can definitely affect their performance at runtime. This section provides guidance on configuring your view objects to get the best possible performance.

35.2.1 Use Bind Variables for Parameterized Queries

Whenever the WHERE clause of your query includes values that might change from execution to execution, you should use named bind variables. The View Criteria Editor that you display from the Query page of the view object's overview editor makes this an easy task. Their use also protects your application against abuse through SQL injection attacks by malicious end-users. For information about defining view criteria with bind variables, see [Section 5.10.1, "How to Create Named View Criteria Declaratively"](#).

35.2.1.1 Use Bind Variables to Avoid Re-parsing of Queries

Bind variables are place holders in the SQL string whose value you can easily change at runtime without altering the text of the SQL string itself. Since the query text doesn't change from execution to execution, the database can efficiently reuse the same parsed statement each time. Avoiding re-parsing of your statement alleviates the database from having to continually re-determine its query optimization plan and eliminates contention by multiple end-users on other costly database resources used during this parsing operation. This savings leads to higher runtime performance of your application. See [Section 5.9.1, "How to Add Bind Variables to a View Object Definition"](#) for details on how to use named bind variables.

35.2.1.2 Use Bind Variables to Prevent SQL-Injection Attacks

Using bind variables for parameterized WHERE clause values is especially important if their values will be supplied by *end-users* of your application. Consider the example shown in [Example 35–3](#). It adds a dynamic WHERE clause formed by concatenating a user-supplied parameter value into the statement.

Example 35–3 Using String Concatenation Instead of Bind Variables is Vulnerable to SQL-Injection Attacks

```
// EXAMPLE OF BAD PRACTICE, Do not follow this approach!
String userSuppliedValue = ... ;
yourViewObject.setWhereClause("BANK_ACCOUNT_ID = "+userSuppliedValue);
```

A user with malicious intentions — if able to learn any details about your application's underlying database schema — could supply a carefully-constructed "bank account number" as a field value or URL parameter like:

BANK_ACCOUNT_ID

When the code in [Example 35–3](#) concatenates this value into the dynamically-applied where clause, what the database sees is a query predicate like this:

WHERE (BANK_ACCOUNT_ID = BANK_ACCOUNT_ID)

This WHERE clause retrieves *all* bank accounts instead of just the current user's, perhaps allowing the hacker to view private information of another person's account. This technique of short-circuiting an application's WHERE clause by trying to supply a maliciously-constructed parameter value into a SQL statement is called a SQL injection

attack. Using named bind variables instead for these situations as shown in [Example 35–4](#) prevents the vulnerability.

Example 35–4 Use Named Bind Variables Instead of String Concatenation

```
// Best practice using named bind variables
String userSuppliedValue = ... ;
yourViewObject.setWhereClause("BANK_ACCOUNT_ID = :BankAcccountId");
yourViewObject.defineNamedWhereClauseParam("BankAcccountId", null, null);
yourViewObject.setNamedWhereClauseParam("BankAcccountId", userSuppliedValue);
```

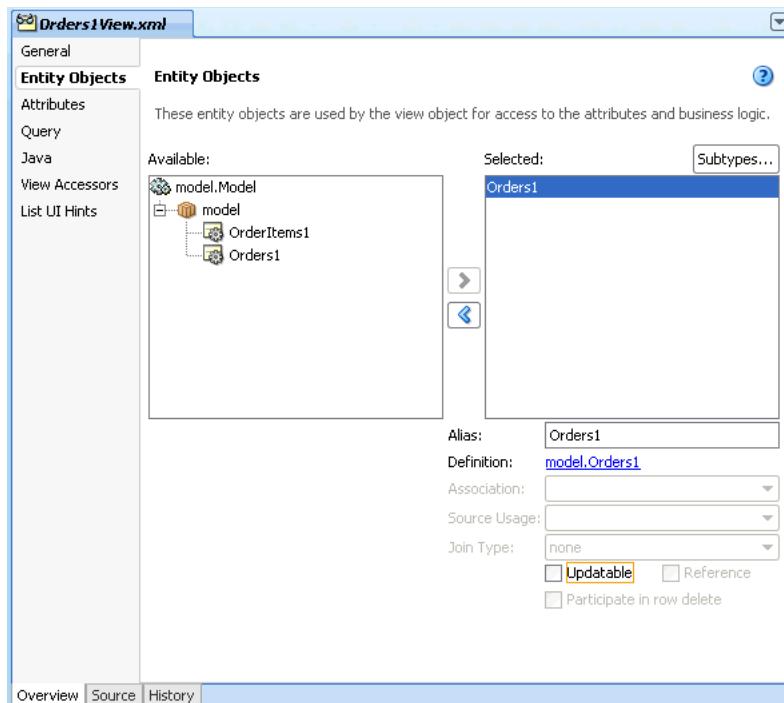
If a malicious user supplies an illegal value in this case, they receive an error your application can handle instead of obtaining data they are not suppose to see.

35.2.2 Consider Using Entity-Based View Objects for Read-Only Data

Typically view objects used for SQL-based validation purposes, as well as for displaying the list of valid selections in a dropdown list, can be read-only. You need to decide what kind of functionality your application requires and design the view object accordingly.

Best Practice: When you need to create a read-only view object for data lookup, you should use the entity-based view object and deselect the **Updatable** option in the view object editor. The approach benefits from the design time editors which aid in generating the SQL query. The alternative of creating an expert-mode view object requires writing a SQL query. Expert mode queries are still useful for cases where Unions and Group By queries cannot be expressed using entity objects.

View objects can either be related to underlying entity objects or not. When a view object is related to one or more underlying entity objects the default behavior supports creating new rows and modifying or removing queried rows. However, the update feature can be disabled by deselecting **Updatable** in the overview editor for the entity-based view object, as shown in [Figure 35–4](#).

Figure 35–4 Deselecting the Updatable Option for an Entity-based View Object

The alternative is to create a read-only view object and define the SQL query using **Expert Mode** in the Edit Query dialog. For the Business Components developer not comfortable with constructing a complex SQL statement, it will always be more convenient to create a non-updatable view object based on an entity object since the editor simplifies the task of creating the query. Entity-based view objects that you set to non-updatable compare favorably to read-only, expert mode-based view objects:

- There is the ability to optimize the select list at runtime to include only those attributes that are required by the user interface
- There is no significant performance degradation incurred by using the entity object to create the local cache
- The data in the view object will reflect the state of the local cache rather than need to return to the database for each read operation
- The data in the local cache will stay consistent should another view object you define need to perform an update on the non-updatable view object's base entity object.

So, while there is a small amount of runtime overhead associated with the coordination between view object rows and entity object rows (estimates show less than 5% overhead), weigh this against the ability to keep the view object definition entirely declarative and maintain a customizable view object. Expert mode-based view objects are not customizable but they can be used to perform Unions and Group By queries that cannot be expressed in entity objects. Expert mode-based view objects are also useful in SQL-based validation queries used by the view object-based Key Exists validator.

When data is not read-only, the best (and only) choice is to create entity-based view objects. Entity-based view objects that are updatable (default behavior) are the only way to pickup entity-derived attribute default values, reflect pending changes made to relevant entity object attributes through other view objects in the same transaction,

and reflect updated reference information when foreign key attribute values are changed is to use an entity-based view object.

35.2.3 Use SQL Tracing to Identify Ill-Performing Queries

After deciding whether your view object should be mapped to entities or not, your attention should turn to the query itself. On the Query page of the view object's overview editor, click the **Edit SQL Query** icon to display the View Object Editor dialog. Click the **Explain Plan** button on the **Query** panel of the View Object Editor to see the query plan that the database query optimizer will use. If you see that it is doing a full table scan, you should consider adding indexes or providing a value for the **Query Optimizer Hint** field on the Tuning section of the overview editor's General page. This will let you explicitly control which query plan will be used. These facilities provide some useful tools to the developer to evaluate the query plans for individual view object SQL statements. However, their use is not a substitute for tracing the SQL of the entire application to identify poorly performing queries in the presence of a production environment's amount of data and number of end users.

You can use the Oracle database's SQL Tracing facilities to produce a complete log of all SQL statements your application performs. The approach that works in all versions of the Oracle database is to issue the command:

```
ALTER SESSION SET SQL_TRACE TRUE
```

Specifically in version 10g of Oracle, the DBA would need to grant `ALTER SESSION` privilege in order to execute this command.

This command enables tracing of the current database session and logs all SQL statements to a server-side trace file until you either enter `ALTER SESSION SET SQL_TRACE FALSE` or close the connection. To simplify enabling this option to trace your Fusion web applications, override the `afterConnect()` method of your application module (or custom application module framework extension class) to conditionally perform the `ALTER SESSION` command to enable SQL tracing based on the presence of a Java system property as shown in [Example 35–5](#).

Example 35–5 Conditionally Enabling SQL Tracing in an Application Module

```
// In YourCustomApplicationModuleImpl.java
protected void afterConnect() {
    super.afterConnect();
    if (System.getProperty("enableTrace") != null) {
        getDBTransaction().executeCommand("ALTER SESSION SET SQL_TRACE TRUE");
    }
}
```

After producing a trace file, you use the `TKPROF` utility supplied with the database to format the information and to better understand information about each query executed like:

- The number of times it was (re)parsed
- The number of times it was executed
- How many round-trips were made between application server and the database
- Various quantitative measurements of query execution time

Using these techniques, you can decide which additional indexes might be required to speed up particular queries your application performs, or which queries could be changed to improve their query optimization plan. For details about working with the

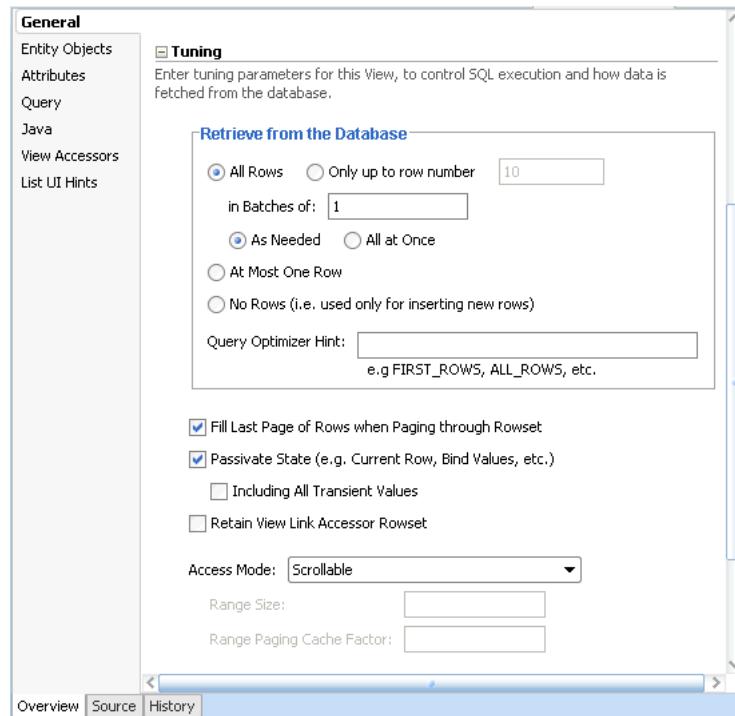
TKPROF utility, see sections "Understanding SQL Trace and TKPROF" and "Using the SQL Trace Facility and TKPROF" in the *Oracle Database Performance Tuning Guide*.

Note: The Oracle database provides the DBMS_MONITOR package that further simplifies SQL tracing and integrates it with Oracle Enterprise Manager for visually monitoring the most frequently performed query statements your applications perform.

35.2.4 Consider the Appropriate Tuning Settings for Every View Object

The Tuning section on the General page of the view object's overview editor lets you set various options that can dramatically effect your query's performance. [Figure 35–5](#) shows the default options that the new view object defines.

Figure 35–5 View Object Default Tuning Options



35.2.4.1 Set the Database Retrieval Options Appropriately

The **Retrieve from the Database** group box, controls how the view object retrieves rows from the database server. The options for the fetch mode are **All Rows**, **Only Up To Row Number**, **At Most One Row**, and **No Rows**. Most view objects will stick with the default **All Rows** option, which will be retrieved **As Needed** (default) or **All at Once** depending on which option you choose.

Note: The **All at Once** option does not enforce a single database round trip to fetch the rows specified by the view object query. The **As Needed** and **All at Once** options work in conjunction with the value of **in Batches of** (also known as fetch size) to determine the number of round trips. For best database access performance, you should consider changing the fetch size as described in [Section 35.2.4.2, "Consider Whether Fetching One Row at a Time is Appropriate"](#).

The **As Needed** option ensures that an `executeQuery()` operation on the view object initially retrieves only as many rows as necessary to fill the first page of a display, whose number of rows is set based on the view object's range size. If you use **As Needed**, then you will require only as many database round trips as necessary to deliver the number of rows specified by the initial range size. Whereas, if you use **All at Once**, then the application will perform as many round trips as necessary to deliver all the rows based on the value of **in Batches of** (fetch size) and the number of rows identified by the query.

For view objects whose `WHERE` clause expects to retrieve a *single* row, set the option to **At Most One Row** for best performance. This way, the view object knows you don't expect any more rows and will skip its normal test for that situation. Finally, if you use the view object only for creating new rows, set the option to **No Rows** so no query will ever be performed.

35.2.4.2 Consider Whether Fetching One Row at a Time is Appropriate

The fetch size controls how many rows will be returned in each round trip to the database. By default, the framework will fetch rows in batches of one row at a time. If you are fetching any more than one row, you will gain efficiency by setting this **in Batches of** value.

However the higher the number, the larger the client-side buffer required, so avoid setting this number arbitrarily high. If you are displaying results N rows at a time in the user interface, it's good 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 the database.

Caution: Unless your query *really* fetches just one row, leaving the *default* fetch size of one (1) in the **in Batches of** field on the Tuning section of the General page of the view object's overview editor is a recipe for bad performance due to many unnecessary round trips between the application server and the database. Oracle strongly recommends considering the appropriate value for each view object's fetch size.

35.2.4.3 Specify a Query Optimizer Hint if Necessary

The **Query Optimizer Hint** field allows you to specify an optional hint to the Oracle query optimizer to influence what execution plan it will use. You can set this hint in the Tuning page of the overview editor for the view object, as shown in [Figure 35–5](#).

At runtime, the hint you provide is added immediately after the `SELECT` keyword in the query, wrapped by the special comment syntax `/*+ YOUR_HINT */`. Two common optimizer hints are:

- `FIRST_ROWS` — to hint that you want the first rows as quickly as possible
- `ALL_ROWS` — to hint that you want all rows as quickly as possible

There are many other optimizer hints that are beyond the scope of this manual to document. Reference the Oracle database reference manuals for more information on available hints.

35.2.5 Creating View Objects at Design Time

It's important to understand the overhead associated with creating view objects at runtime. Avoid the temptation to do this without a compelling business requirement. For example, if your application issues a query against a table whose name you know at design time and if the list of columns to retrieve is also fixed, then create a view object at design time. When you do this, your SQL statements are neatly encapsulated, can be easily explained and tuned during development, and incur no runtime overhead to discover the structure and data types of the resulting rows.

In contrast, when you use the `createViewObjectFromQueryStmt()` API on the `ApplicationModule` interface at runtime, your query is buried in code, it's more complicated to proactively tune your SQL, and you pay a performance penalty each time the view object is created. Since the SQL query statement for a dynamically-created view object could theoretically be different each time a new instance is created using this API, an extra database round trip is required to discover the "shape" of the query results on-the-fly. Only create queries dynamically if you *cannot* know the name of the table to query until runtime. Most other needs can be addressed using a design-time created view object in combination with runtime API's to set bind variables in a fixed where clause, or to add an additional WHERE clause (with optional bind variables) at runtime.

35.2.6 Use Forward Only Mode to Avoid Caching View Rows

Often you will write code that programmatically iterates through the results of a view object. A typical situation will be custom validation code that must process multiple rows of query results to determine whether an attribute or an entity is valid or not. In these cases, if you intend to read each row in the row set a single time and never require scrolling backward or re-iterating the row set a subsequent time, then you can use "forward only" mode to avoid caching the retrieved rows. To enable forward only mode, call `setForwardOnly(true)` on the view object.

Note: Using a read-only view object (with no entity usages) in forward-only mode with an appropriately tuned fetch size is the most efficient way to programmatically read data.

You can also use forward-only mode to avoid caching rows when inserting, updating, or deleting data as long as you never scroll backward through the row set and never call `reset()` to set the iterator back to the first row. Forward only mode only works with a range size of one (1).

35.3 Using Expert Mode for Full Control Over SQL Query

Although JDeveloper provides declarative support for many SQL features, when defining entity-based view objects, you still have the option to fully-specify the `WHERE` and `ORDER BY` clauses, whereas, by default, the `FROM` clause and `SELECT` list are automatically derived. When you require full control over the `SELECT` or `FROM` clause in a query, you can enable "expert mode" in either the wizard or overview editor for the view object. In expert mode, the names of the tables related to the participating entity usages determine the `FROM` clause, while the `SELECT` list is based on the:

- Underlying column names of participating entity-mapped attributes
- SQL expressions of SQL-calculated attributes

Tips: The view object editors and wizard in the JDeveloper provide full support for generating SQL from choices that you make. For example, two such options allow you to declaratively define outer joins and work in declarative SQL mode (where no SQL is generated until runtime). For details about working declaratively with view objects, see [Chapter 5, "Defining SQL Queries Using View Objects"](#).

35.3.1 How to Enable Expert Mode for Full SQL Control

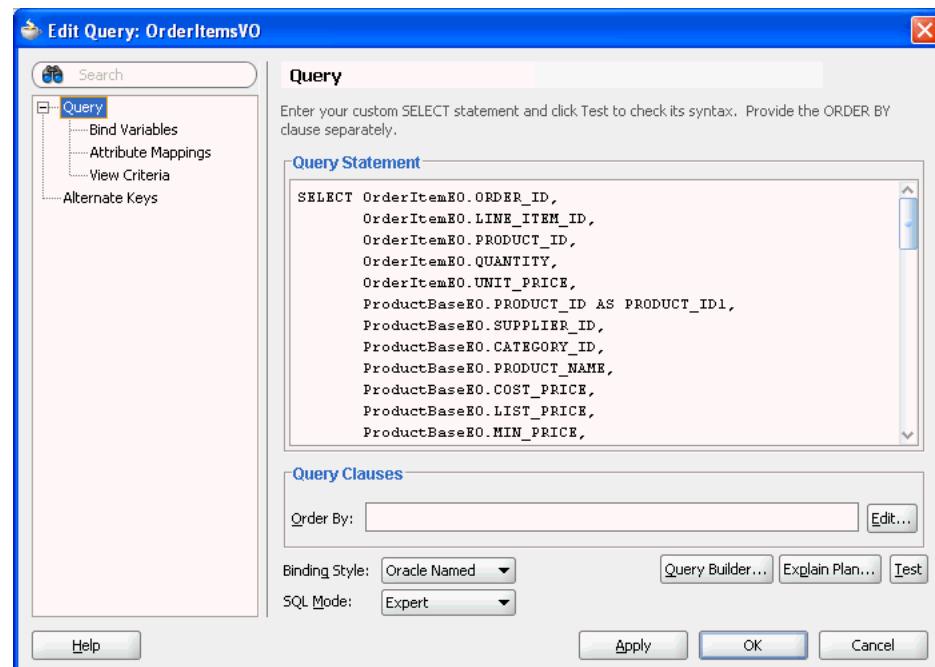
To enable expert mode, select **Expert Mode** on the Query panel of the Create View Object wizard. You can also modify the SQL statement of an existing entity-based view object in the view object overview editor. In the overview editor, navigate to the Query section and click the **Edit SQL Query** button. In the Edit Query dialog, select **Expert Mode** from the **SQL Mode** dropdown list.

35.3.2 What Happens When You Enable Expert Mode

When you enable expert mode, the read-only Generated Statement section of the SQL Statement panel becomes a fully-editable Query Statement text box, displaying the full SQL statement. Using this text box, you can change every aspect of the SQL query.

For example, [Figure 35–6](#) shows the Query page of the Edit Query dialog for the Fusion Order Demo application's OrderItemsVO view object. It's an expert mode, entity-based view object that joins various tables.

Figure 35–6 OrderItemsVO Expert Mode View Object in the Fusion Order Demo Application



35.3.3 What You May Need to Know

When you work with the modal view object editor to create an expert mode SQL statement, be sure to observe the following guidelines.

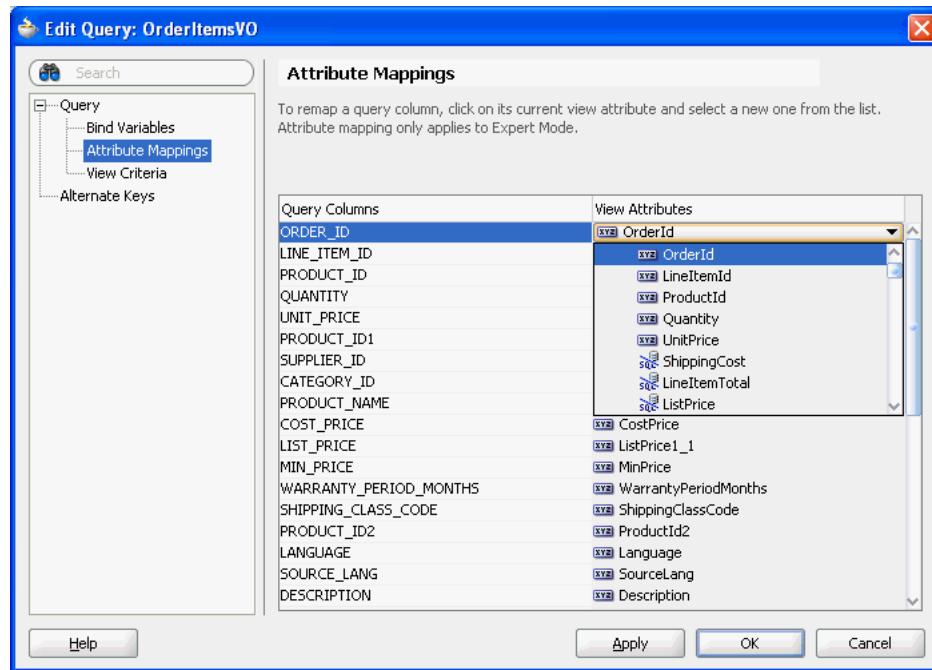
35.3.3.1 You May Need to Perform Manual Attribute Mapping

The automatic cooperation of a view object with its underlying entity objects depends on correct attribute-mapping metadata saved in the XML component definition. This information relates the view object attributes to corresponding attributes from participating entity usages. JDeveloper maintains this attribute mapping information in a fully-automatic way for normal entity-based view objects. However, when you decide to use expert mode with a view object, you need to pay attention to the changes you make to the SELECT list. That is the part of the SQL query that directly relates to the attribute mapping. Even in expert mode, JDeveloper continues to offer some assistance in maintaining the attribute mapping metadata when you do the following to the SELECT list:

- Reorder an expression without changing its column alias
JDeveloper reorders the corresponding view object attribute and maintains the attribute mapping.
- Add a new expression
JDeveloper adds a new SQL-calculated view object attribute with a corresponding Camel-Capped name based on the column alias of the new expression.
- Remove an expression
JDeveloper converts the corresponding SQL-calculated or entity-mapped attribute related to that expression to a transient attribute.

However, if you rename a column alias in the SELECT list, JDeveloper has no way to detect this, so it is treated as if you removed the old column expression and added a new one of a different name.

After making any changes to the SELECT list of the query, click the **Edit SQL Query** button in the overview editor's Query section and visit the **Attribute Mappings** panel of the Edit Query dialog to ensure that the attribute mapping metadata is correct. The table in this panel, which is disabled for view objects in normal mode, becomes enabled for expert mode view objects, as shown in [Figure 35-4](#). For each view object attribute, you will see its corresponding SQL column alias in the table. By clicking in a cell in the **View Attributes** column, you can use the dropdown list that appears to select the appropriate entity object attribute to which any entity-mapped view attributes should correspond.

Figure 35–7 OrderItemsVO Attribute Mappings in the Fusion Order Demo Application

Note: If the view attribute is SQL-calculated or transient, the Attribute Mappings panel displays the attribute with a SQL icon in the **View Attributes** column to represent it. Since these attributes are not related to underlying entity objects, there is no entity attribute related information required for them.

35.3.3.2 Disabling Expert Mode Loses Any Custom Edits

When you disable expert mode for a view object it will return to having its SELECT and FROM clause be derived again. JDeveloper warns you that doing this might lose any of your custom edits to the SQL statement. If this is what you want, after acknowledging the alert, your view object's SQL query reverts back to the default.

35.3.3.3 Once In Expert Mode, Changes to SQL Expressions Are Ignored

Consider a Products view object with a SQL-calculated attribute named Shortens whose SQL expression you defined as SUBSTR (NAME , 1 , 10) . If you switch this view object to expert mode, the **Query Statement** box will show a SQL query like this:

```
SELECT Products.PROD_ID,
       Products.NAME,
       Products.IMAGE,
       Products.DESCRIPTION,
       SUBSTR(NAME,1,10) AS SHORT_NAME
  FROM PRODUCTS Products
```

If you go back to the attribute definition for the Shortens attribute and change the **SQL Expression** field from SUBSTR (NAME , 1 , 10) to SUBSTR (NAME , 1 , 15) , then the change will be saved in the view object's XML component definition. Note, however, that the SQL query will remain as above. This occurs because JDeveloper never tries to *modify* the text of an expert mode query. In expert mode, the developer is in full control. JDeveloper attempts to adjust metadata as described above in function of

some kinds of changes you make yourself to the expert mode SQL statement, but it does not perform the reverse. Therefore, if you change view object metadata, the expert mode SQL statement is not updated to reflect it.

To make the above change to the SQL calculated `Shortens` attribute, you need to update the expression in the expert mode SQL statement itself. To be 100% thorough, you should make the change *both* in the attribute metadata *and* in the expert mode SQL statement. This would ensure — if you (or another developer on your team) ever decides to toggle expert mode *off* at a later point in time — that the automatically derived SELECT list would contain the correct SQL-derived expression.

Note: If you find you had to make numerous changes to the view object metadata of an expert mode view object, you can consider the following technique to avoid having to manually translate any effects those changes might have implied to the SQL statement yourself.

First, copy the text of your customized query to a temporary file. Then, disable expert mode for the view object and acknowledge the warning that you will lose your changes. At this point JDeveloper will re-derive the correct generated SQL statement based on all the new metadata changes you've made. Finally, you can enable expert mode once again and re-apply your SQL customizations.

35.3.3.4 Don't Map Incorrect Calculated Expressions to Entity Attributes

When changing the SELECT list expression that corresponds to *entity-mapped* attributes, don't introduce SQL calculations that change the value of the attribute when retrieving the data. To illustrate the problem that will occur if you do this, consider the following query for a simple entity-based view object named `Products`:

```
SELECT Products.PROD_ID,
       Products.NAME,
       Products.IMAGE,
       Products.DESCRIPTION
  FROM PRODUCTS Products
```

Imagine that you wanted to limit the name column to showing only the first ten characters of the name for some use case. The correct way to do that would be to introduce a new SQL-calculated field called `ShortName` with an expression like `SUBSTR(Products.NAME, 1, 10)`. However, one way you might have thought to accomplish this was to switch the view object to expert mode and change the SELECT list expression for the entity-mapped `NAME` column to the following:

```
SELECT Products.PROD_ID,
       SUBSTR(Products.NAME, 1, 10) AS NAME,
       Products.IMAGE,
       Products.DESCRIPTION
  FROM PRODUCTS Products
```

This alternative strategy would initially appear to work. At runtime, you see the truncated value of the name as you are expecting. However, if you modify the row, when the underlying entity object attempts to lock the row it does the following:

- Issues a `SELECT FOR UPDATE` statement, retrieving all columns as it tries to lock the row.
- If the entity object successfully locks the row, it compares the original values of all the persistent attributes in the entity cache as they were last retrieved from the

database with the values of those attributes just retrieved from the database during the lock operation.

- If any of the values differs, then the following error is thrown:

```
(oracle.jbo.RowInconsistentException)
JBO-25014: Another user has changed the row with primary key [...]
```

If you see an error like this at runtime even though you are the *only* user testing the system, it is most likely due to your inadvertently introducing a SQL function in your expert mode view object that changed the selected value of an entity-mapped attribute. In the example above, the SUBSTR(Products.NAME, 1, 10) function introduced causes the original selected value of the Name attribute to be truncated. When the row-lock SQL statement selects the value of the NAME column, it will select the entire value. This will cause the comparison described above to fail, producing the "phantom" error that another user has changed the row.

The same thing would happen with NUMBER, or DATE valued attributes if you inadvertently apply SQL functions in expert mode to truncate or alter their retrieved values for entity-mapped attributes. If you need to present altered versions of entity-mapped attribute data, introduce a new SQL-calculated attribute with the appropriate expression to handle the job.

35.3.3.5 Expert Mode SQL Formatting is Retained

When you switch a view object to expert mode, its XML component definition switches from storing parts of the query in separate XML attributes, to saving the entire query in a single <SQLQuery> element. The query is wrapped in a XML CDATA section to preserve the line formatting you may have done to make a complex query be easier to understand.

35.3.3.6 Expert Mode Queries Are Wrapped as Inline Views

If your expert-mode view object:

- Contains a design-time ORDER BY clause specified in the **Order By** field of the **Query Clauses** group box, or
- Has a dynamic WHERE clause or order by clause applied at runtime using setWhereClause() or setOrderByClause()

then its query gets nested into an inline view before applying these clauses. For example, suppose your expert-mode query was defined as:

```
select CUSTOMER_ID, EMAIL, FIRST_NAME, LAST_NAME
from CUSTOMERS
union all
select CUSTOMER_ID, EMAIL, FIRST_NAME, LAST_NAME
from INACTIVE_CUSTOMERS
```

At runtime, when you set an additional WHERE clause like email = :TheUserEmail, the view object nests its original query into an inline view like this:

```
SELECT * FROM(
select CUSTOMER_ID, EMAIL, FIRST_NAME, LAST_NAME
from CUSTOMERS
union all
select CUSTOMER_ID, EMAIL, FIRST_NAME, LAST_NAME
from INACTIVE_CUSTOMERS) QRSLT
```

and then adds the dynamic where clause predicate at the end, so that the final query the database sees is:

```
SELECT * FROM(
select CUSTOMER_ID, EMAIL, FIRST_NAME, LAST_NAME
from CUSTOMERS
union all
select CUSTOMER_ID, EMAIL, FIRST_NAME, LAST_NAME
from INACTIVE_CUSTOMERS) QRSLT
WHERE email = :TheCustomerEmail
```

This query "wrapping" is necessary in general for expert mode queries since the original query could be arbitrarily complex, including SQL UNION, INTERSECT, MINUS, or other operators that combine multiple queries into a single result. In those cases, simply "gluing" the additional runtime WHERE clause onto the end of the query text could produce unexpected results since. For example, it might only apply to the *last* of several UNION'ed statements. By nesting the original query verbatim into an inline view, the view object guarantees that your additional WHERE clause is correctly used to filter the results of the original query, regardless of how complex it is.

35.3.3.7 Disabling the Use of Inline View Wrapping at Runtime

Due to the inline view wrapping of expert mode view objects, the dynamically-added WHERE clause can only refer to columns in the SELECT list of the original query. To avoid this limitation, when necessary you can disable the use of the inline view wrapping by calling `setNestedSelectForFullSql(false)`, typically in an overridden `create()` method of your view object.

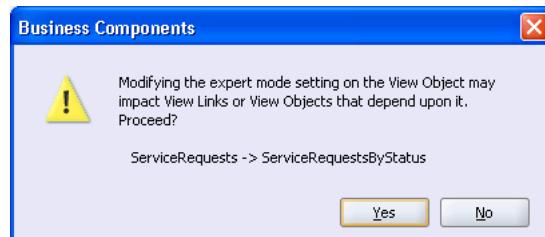
35.3.3.8 Enabling Expert Mode May Impact Dependent Objects

When you modify a query to be in expert mode after you have already created:

- View links involving it, or
- Other view objects that extend it

JDeveloper will warn you with the alert shown in [Figure 35–8](#) to remind you that you should revisit these dependent components to ensure their SQL statements still reflect the correct query.

Figure 35–8 Proactive Reminder to Revisit Dependent Components



For example, if you were to modify a `Orders` view object to use expert mode, and assuming the `OrdersStatus` view object extends it, you would need to revisit the extended component to ensure its query still logically reflects an extension of the modified parent component.

35.4 Generating Custom Java Classes for a View Object

As you've seen, all of the basic querying functionality of a view object can be achieved without using custom Java code. Clients can retrieve and iterate through the data of any SQL query without resorting to any custom code on the view object developer's part. In short, for many read-only view objects, once you have defined the SQL statement, you're done. However, it's important to understand how to enable custom Java generation for a view object when your needs might require it. For example, reasons you might write code in a custom Java class include:

- To add validation methods (although Groovy Script expressions can provide this support without needing Java)
- To add custom logic
- To augment built-in behavior

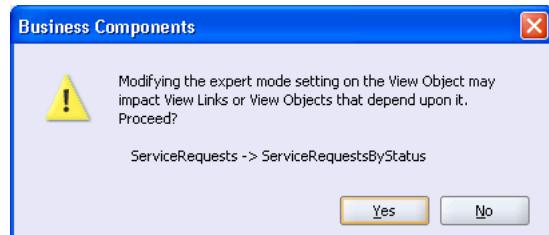
[Appendix E, "Most Commonly Used ADF Business Components Methods"](#) provides a quick reference to the most common code that you will typically write, use, and override in your custom view object and view row classes.

35.4.1 How To Generate Custom Classes

To enable the generation of custom Java classes for a view object, use the **Java** page of the View Object Editor. As shown in [Figure 35–9](#), there are three optional Java classes that can be related to a view object. The first two in the list are the most commonly used:

- *View object class*, which represents the component that performs the query
- *View row class*, which represents each row in the query result

Figure 35–9 View Object Custom Java Generation Options



35.4.1.1 Generating Bind Variable Accessors

When you enable the generation of a custom view object class, if you also select the **Bind Variable Accessors** checkbox, then JDeveloper generates getter and setter methods in your view object class. Since the `Users` view object had three named bind variables (`TheName`, `LowUserId`, and `HighUserId`), the custom `UsersImpl.java` view object class would have corresponding methods like this:

```
public Number getLowUserId() {...}
public void setLowUserId(Number value) {...}
public Number getHighUserId() {...}
public void setHighUserId(Number value) {...}
public String getTheName() {...}
public void setTheName(String value) {...}
```

These methods allow you to set a bind variable with compile-time type-checking to ensure you are setting a value of the appropriate type. That is, instead of writing a line like this to set the value of the `LowUserId`:

```
vo.setNamedWhereClauseParam("LowUserId", new Number(150));
```

You can write the code like:

```
vo.setLowUserId(new Number(150));
```

You can see that with the latter approach, the Java compiler would catch a typographical error had you accidentally typed `setLowUserName` instead of `setLowUserId`:

```
// spelling name wrong gives compile error
vo.setLowUserName(new Number(150));
```

Or if you were to incorrectly pass a value of the wrong data type, like "ABC" instead of `Number` value:

```
// passing String where number expected gives compile error
vo.setLowUserId("ABC");
```

Without the generated bind variable accessors, an incorrect line of code like the following cannot be caught by the compiler:

```
// Both variable name and value wrong, but compiler cannot catch it
vo.setNamedWhereClauseParam("LowUserName", "ABC");
```

It contains both an incorrectly spelled bind variable name, as well as a bind variable value of the wrong datatype. If you use the generic APIs on the `ViewObject` interface, errors of this sort will raise exceptions at runtime instead of being caught at compile time.

35.4.1.2 Generating View Row Attribute Accessors

When you enable the generation of a custom view row class, if you also select the **Accessors** checkbox, then JDeveloper generates getter and setter methods for each attribute in the view row. For example, for the `Users` view object, the corresponding custom `UsersRowImpl.java` class might have methods like this generated in it:

```
public Number getUserId() {...}
public void setId(Number value) {...}
public String getEmail() {...}
public void setEmail(String value) {...}
public String getFirstName() {...}
public void setFirstName(String value) {...}
public String getLastName() {...}
public void setLastName(String value) {...}
public String getUserRole() {...}
public void setUserRole(String value) {...}
```

These methods allow you to work with the row data with compile-time checking of the correct datatype usage. That is, instead of writing a line like this one that gets the value of the `UserId` attribute:

```
Number userId = (Number)row.getAttribute("UserId");
```

you can write the code like:

```
Number userId = row.getUserId();
```

You can see that with the latter approach, the Java compiler would catch a typographical error had you accidentally typed `UserIdentifier` instead of `UserId`:

```
// spelling name wrong gives compile error
Number userId = row.getUserIdentifier();
```

Without the generated view row accessor methods, an incorrect line of code like the following cannot be caught by the compiler:

```
// Both attribute name and type cast are wrong, but compiler cannot catch it
String userId = (String)row.getAttribute("UserIdentifier");
```

It contains both an incorrectly spelled attribute name, as well as an incorrectly-typed cast of the `getAttribute()` return value. Using the generic APIs on the Row interface, errors of this kind will raise exceptions at runtime instead of being caught at compile time.

35.4.1.3 Exposing View Row Accessors to Clients

When enabling the generation of a custom view row class, if you choose to generate the view row attribute accessor, you can also optionally select the **Expose Accessor to the Client** checkbox. This causes an additional custom row interface to be generated which application clients can use to access custom methods on the row without depending directly on the implementation class. As described in [Section 3.5.9, "Custom Interface Support for Client-Accessible Components"](#), having client code work with business service tier interfaces instead of concrete classes is a best practice to ensure that client code does not need to change when your server-side implementation does.

For example, in the case of the Users view object, exposing the accessors to the client will generate a custom row interface named `UsersRow`. This interface is created in the common subpackage of the package in which the view object resides. Having the row interface allows clients to write code that accesses the attributes of query results in a strongly typed manner. [Example 35–6](#) shows a `TestClient3` sample client program that casts the results of the `next()` method to the `UsersRow` interface so that it can call accessors like `getUserId()` and `getEmail()`.

Example 35–6 Simple Example of Using Client Row Interface with Accessors

```
package devguide.examples.client;
import devguide.examples.common.UsersRow;
import oracle.jbo.ApplicationModule;
import oracle.jbo.ViewObject;
import oracle.jbo.client.Configuration;
import oracle.jbo.domain.Number;
public class TestClient3 {
    public static void main(String[] args) {
        String amDef = "devguide.examples.UserService";
        String config = "UserServiceLocal";
        ApplicationModule am =
            Configuration.createRootApplicationModule(amDef, config);
        ViewObject vo = am.findViewObject("UserList");
        vo.executeQuery();
        while (vo.hasNext()) {
            // Cast next() to a strongly-typed UsersRow interface
            UsersRow curUser = (UsersRow)vo.next();
            Number userId = curUser.getUserId();
            String email = curUser.getEmail();
            System.out.println(userId+ " " + email);
        }
    }
}
```

```

        Configuration.releaseRootApplicationModule(am, true);
    }
}

```

35.4.1.4 Configuring Default Java Generation Preferences

You've seen how to generate custom Java classes for your view objects when you need to customize their runtime behavior, or if you simply prefer to have strongly typed access to bind variables or view row attributes.

To change the default settings that control how JDeveloper generates Java classes, choose **Tools | Preferences** and open the **Business Components** page. The settings you choose will apply to all future business components you create.

Oracle recommends that developers getting started with ADF Business Components set their preference to generate no custom Java classes by default. As you run into specific needs, you can enable just the bit of custom Java you need for that one component. Over time, you'll discover which set of defaults works best for you.

35.4.2 What Happens When You Generate Custom Classes

When you choose to generate one or more custom Java classes, JDeveloper creates the Java file(s) you've indicated.

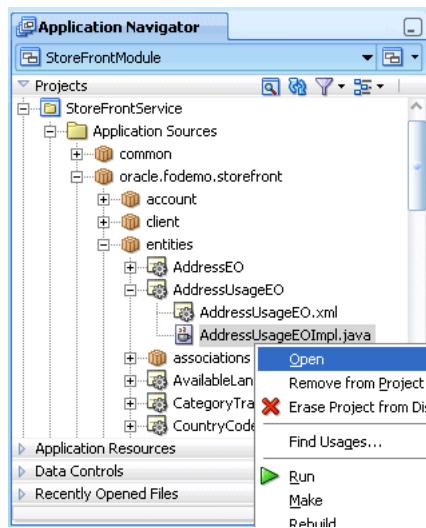
For example, in the case of a view object named `devguide.examples.Users`, the default names for its custom Java files will be `UsersImpl.java` for the view object class and `UsersRowImpl.java` for the view row class. Both files get created in the same `./devguide/examples` directory as the component's XML component definition file.

The Java generation options for the view object continue to be reflected on the Java page on subsequent visits to the View Object Editor. Just as with the XML definition file, JDeveloper keeps the generated code in your custom Java classes up to date with any changes you make in the editor. If later you decide you didn't require a custom Java file for any reason, unchecking the relevant options in the Java page causes the custom Java files to be removed.

35.4.2.1 Seeing and Navigating to Custom Java Files

As with all ADF components, when you select a view object in the Application Navigator, the Structure window displays all of the implementation files that comprise it. The only required file is the XML component definition file. You saw above that when translatable UI control hints are defined for a component, it will have a component message bundle file as well. As shown in [Figure 35-10](#), when you've enabled generation of custom Java classes, they also appear under the **Sources** folder for the view object. When you need to see or work with the source code for a custom Java file, there are two ways to open the file in the source editor:

- Choose the relevant **Go to** option in the context menu as shown in [Figure 35-10](#)
- Double-click on a file in the **Sources** folder in the Structure window

Figure 35–10 Seeing and Navigating to Custom Java Classes for a View Object

35.4.3 What You May Need to Know About Custom Classes

See the following sections for additional information to help you use custom Java classes.

35.4.3.1 About the Framework Base Classes for a View Object

When you use an "XML-only" view object, at runtime its functionality is provided by the default ADF Business Components implementation classes. Each custom Java class that gets generated will automatically extend the appropriate ADF Business Components base class so that your code inherits the default behavior and can easily add or customize it. A view object class will extend `ViewObjectImpl`, while the view row class will extend `ViewRowImpl` (both in the `oracle.jbo.server` package).

35.4.3.2 You Can Safely Add Code to the Custom Component File

Based perhaps on previous negative experiences, some developers are hesitant to add their own code to generated Java source files. Each custom Java source code file that JDeveloper creates and maintains for you includes the following comment at the top of the file to clarify that it is safe to add your own custom code to this file:

```
// -----
// --- File generated by Oracle ADF Business Components Design Time.
// --- Custom code may be added to this class.
// --- Warning: Do not modify method signatures of generated methods.
// -----
```

JDeveloper does not blindly regenerate the file when you click the **OK** or **Apply** button in the component editor. Instead, it performs a smart update to the methods that it needs to maintain, leaving your own custom code intact.

35.4.3.3 Attribute Indexes and InvokeAccessor Generated Code

The view object is designed to function either in an XML-only mode or using a combination of an XML component definition and a custom Java class. Since attribute values are not stored in private member fields of a view row class, such a class is not present in the XML-only situation. Instead, in addition to a name, attributes are also assigned a numerical index in the view object's XML component definition, on a

zero-based, sequential order of the <ViewAttribute> and association-related <ViewLinkAccessor> tags in that file. At runtime, the attribute values in an view row are stored in a structure that is managed by the base ViewRowImpl class, indexed by the attribute's numerical position in the view object's attribute list.

For the most part this private implementation detail is unimportant. However, when you enable a custom Java class for your view row, this implementation detail is related to some of the generated code that JDeveloper automatically maintains in your view row class, and you may want to understand what that code is used for. For example, in the custom Java class for the Users view row, [Example 35–7](#) shows that each attribute or view link accessor attribute has a corresponding generated integer constant. JDeveloper ensures that the values of these constants correctly reflect the ordering of the attributes in the XML component definition.

Example 35–7 Attribute Constants Are Automatically Maintained in the Custom View Row Java Class

```
public class UsersRowImpl extends ViewRowImpl implements UsersRow {
    public static final int USERID = 0;
    public static final int EMAIL = 1;
    public static final int FIRSTNAME = 2;
    public static final int LASTNAME = 3;
    public static final int USERROLE = 4;
    public static final int ASSIGNEDREQUESTS = 5;
    // etc.
```

You'll also notice that the automatically maintained, strongly typed getter and setter methods in the view row class use these attribute constants like this:

```
// In devguide.examples.UsersRowImpl class
public String getEmail() {
    return (String) getAttributeInternal(EMAIL); // <-- Attribute constant
}
public void setEmail(String value) {
    setAttributeInternal(EMAIL, value); // <-- Attribute constant
}
```

The last two aspects of the automatically maintained code related to view row attribute constants are the `getAttrInvokeAccessor()` and `setAttrInvokeAccessor()` methods. These methods optimize the performance of attribute access by numerical index, which is how generic code in the `ViewRowImpl` base class typically accesses attribute values. An example of the `getAttrInvokeAccessor()` method looks like the following from the `ServiceRequestImpl.java` class. The companion `setAttrInvokeAccessor()` method looks similar.

```
// In devguide.examples.UsersRowImpl class
protected Object getAttrInvokeAccessor(int index, AttributeDefImpl attrDef)
throws Exception {
    switch (index) {
        case USERID:           return getUserId();
        case EMAIL:            return getEmail();
        case FIRSTNAME:        return getFirstName();
        case LASTNAME:          return getLastName();
        case USERROLE:          return getUserRole();
        case ASSIGNEDREQUESTS: return getAssignedRequests();
        default:
            return super.getAttrInvokeAccessor(index, attrDef);
    }
}
```

The rules of thumb to remember about this generated attribute-index related code are the following.

The Do's

- Add custom code if needed inside the strongly typed attribute getter and setter methods
- Use the View Object Editor to change the order or type of view object attributes JDeveloper will change the Java signature of getter and setter methods, as well as the related XML component definition for you.

The Don'ts

- Don't modify the `getAttrInvokeAccessor()` and `setAttrInvokeAccessor()` methods
- Don't change the values of the attribute index numbers by hand

Note: If you need to manually edit the generated attribute constants, perhaps due to source control merge conflicts, you must ensure that the zero-based ordering reflects the sequential ordering of the `<ViewAttribute>` and `<ViewLinkAccessor>` tags in the corresponding view object XML component definition.

35.5 Working Programmatically with Multiple Named View Criteria

You can define *multiple* named view criteria in the overview editor for a view object and then selectively apply any combination of them to your view object at runtime as needed. For information about working with named view criteria at design time, see [Section 5.10.1, "How to Create Named View Criteria Declaratively"](#).

35.5.1 Applying One or More Named View Criteria

To apply one or more named view criteria, use the `setApplyViewCriteriaNames()` method. This method accepts a String array of the names of the criteria you want to apply. If you apply more than one named criteria, they are AND-ed together in the WHERE clause produced at runtime. New view criteria that you apply with the `setApplyViewCriteriaNames()` method will not overwrite those that were previously applied.

When you need to apply more than one named view criteria, you can expose custom methods on the client interface of the view object to encapsulate applying combinations of the named view criteria. For example, [Example 35–8](#) shows custom methods `showStaffInUS()`, `showCustomersOutsideUS()`, and `showCustomersInUS()`, each of which uses the `setApplyViewCriteriaNames()` method to apply an appropriate combination of named view criteria. Once these methods are exposed on the view object's client interface, at runtime clients can invoke these methods as needed to change the information displayed by the view object.

Example 35–8 Exposing Client Methods to Enable Appropriate Named Criterias

```
// In UsersImpl.java
public void showStaffInUS() {
    setApplyViewCriteriaNames(new String[]{"CountryIsUS", "IsStaff"});
    executeQuery();
}
public void showCustomersOutsideUS() {
```

```

        setApplyViewCriteriaNames(new String[]{"CountryIsNotUS", "IsCustomer"});
        executeQuery();
    }
    public void showCustomersInUS() {
        setApplyViewCriteriaNames(new String[]{"CountryIsUS", "IsCustomer"});
        executeQuery();
    }
}

```

35.5.2 Removing All Applied Named View Criteria

To remove any currently applied named view criteria, use `setApplyViewCriteriaNames(null)`. For example, you could add the `showAll()` method in [Example 35–9](#) to the `Users` view object and expose it on the client interface. This would allow clients to return to an unfiltered view of the data when needed.

Oracle does not recommend removing the design time view criteria because the row level bind variable values may already be applied on the row set. To help ensure this, named view criteria that get defined for a view accessor in the design time, will be applied as "required" view criteria on the view object instance so that it does not get removed by the view criteria's life cycle methods.

Example 35–9 Removing All Applied Named View Criteria

```

// In UsersImpl.java
public void showAll() {
    setApplyViewCriteriaNames(null);
    executeQuery();
}

```

Note: The `setApplyViewCriteriaNames(null)` removes all applied view criteria, but allows you to later reapply any combination of them. In contrast, the `clearViewCriteriaNames()` method *deletes* all named view criteria. After calling `clearViewCriteriaNames()` you would have to use `putViewCriteria()` again to define new named criteria before you could apply them.

35.5.3 Using the Named Criteria at Runtime

[Example 35–10](#) shows the interesting lines of a `TestClient` class that works with the `Users` view object described above. It invokes different client methods on the `Users` view object interface to show different filtered sets of data. The `showRows()` method is a helper method that iterates over the rows in the view object to display some attributes.

Example 35–10 Test Client Code Working with Named View Criterias

```

// In TestClientMultipleViewCriterias.java
Persons vo = (Persons)am.findViewObject("Persons");
vo.showMaleCustomers();
showRows(vo,"After applying view criterias for male customers");
vo.showStaffInUS();
showRows(vo,"After applying view criterias for female staff");
vo.showCustomersInUS();
showRows(vo,"After applying view criterias for female customers");
vo.showAll();
showRows(vo,"After clearing all view criterias");

```

Running the TestClient program produces output as follows:

```
--- After applying view criterias for male customers ---
Hermann Baer [user, DE]
John Chen [user, TH]
:
--- After applying view criterias for female staff ---
David Austin [technician, US]
Bruce Ernst [technician, US]
:
--- After applying view criterias for female customers ---
Shelli Baida [user, US]
Emerson Clabe [user, US]
:
--- After clearing all view criterias ---
David Austin [technician, US]
Hermann Baer [user, DE]
:
```

35.6 Performing In-Memory Sorting and Filtering of Row Sets

By default a view object performs its query against the database to retrieve the rows in its resulting row set. However, you can also use view objects to perform in-memory searches and sorting to avoid unnecessary trips to the database.

35.6.1 Understanding the View Object's SQL Mode

The view object's SQL mode controls the source used to retrieve rows to populate its row set. The `setQueryMode()` allows you to control which mode, or combination of modes, are used:

- `ViewObject.QUERY_MODE_SCAN_DATABASE_TABLES`
This is the default mode that retrieves results from the database.
- `ViewObject.QUERY_MODE_SCAN_VIEW_ROWS`
This mode uses rows already in the row set as the source, allowing you to progressively refine the row set's contents through in-memory filtering.
- `ViewObject.QUERY_MODE_SCAN_ENTITY_ROWS`
This mode, valid only for entity-based view objects, uses the entity rows presently in the entity cache as the source to produce results based on the contents of the cache.

You can use the modes individually, or combine them using Java's logical OR operator (`X | Y`). For example, to create a view object that queries the entity cache for unposted new entity rows, as well as the database for existing rows, you could write code like:

```
setQueryMode(ViewObject.QUERY_MODE_SCAN_DATABASE_TABLES |
            ViewObject.QUERY_MODE_SCAN_ENTITY_ROWS)
```

If you combine the SQL modes, the view object automatically handles skipping of duplicate rows. In addition, there is an implied order to the results that are found:

1. Scan view rows (if specified)
2. Scan entity cache (if specified)
3. Scan database tables (if specified) by issuing a SQL query

If you call the `setQueryMode()` method to change the SQL mode, your new setting takes effect the next time you call the `executeQuery()` method.

35.6.2 Sorting View Object Rows In Memory

To sort the rows in a view object at runtime, use the `setSortBy()` method. You pass a sort expression that looks like a SQL ORDER BY clause. However, instead of referencing the column names of the table, you use the view object's attribute names. For example, for a view object containing attributes named `Customer` and `DaysOpen`, you could sort the view object first by `Customer` descending, then by `DaysOpen` by calling:

```
setSortBy("Customer desc, DaysOpen");
```

Alternatively, you can use the *zero-based* attribute index position in the sorting clause like this:

```
setSortBy("3 desc, 2");
```

After calling the `setSortBy()` method, the rows will be sorted the next time you call the `executeQuery()` method. The view object translates this sorting clause into an appropriate format to use for ordering the rows depending on the SQL mode of the view object. If you use the default SQL mode, the `SortBy` clause is translated into an appropriate ORDER BY clause and used as part of the SQL statement sent to the database. If you use either of the in-memory SQL modes, then the `SortBy` by clause is translated into one or more `SortCriteria` objects for use in performing the in-memory sort.

Note: While SQL ORDER BY expressions treat column names in a case-insensitive way, the attribute names in a `SortBy` expression are case-sensitive.

35.6.2.1 Combining `setSortBy` and `setQueryMode` for In-Memory Sorting

[Example 35–11](#) shows the interesting lines of code from the `TestClientSetSortBy` class that uses `setSortBy()` and `setQueryMode()` to perform an in-memory sort on the rows produced by a read-only view object `ClosedOrders`.

Example 35–11 Combining `setSortBy` and `setQueryMode` for In-Memory Sorting

```
// In TestClientSetSortBy.java
am.getTransaction().executeCommand("ALTER SESSION SET SQL_TRACE TRUE");
ViewObject vo = am.findViewObject("ClosedOrders");
vo.executeQuery();
showRows(vo, "Initial database results");
vo.setSortBy("Customer desc");
vo.setQueryMode(ViewObject.QUERY_MODE_SCAN_VIEW_ROWS);
vo.executeQuery();
showRows(vo, "After in-memory sorting by Customer desc");
vo.setSortBy("Customer desc, DaysOpen");
vo.executeQuery();
showRows(vo, "After in-memory sorting by Customer desc, DaysOpen");
```

The first line in [Example 35–11](#) containing the `executeCommand()` call issues the `ALTER SESSION SET SQL TRACE` command to enable SQL tracing for the current database session. This causes the Oracle database to log every SQL statement performed to a server-side trace file. It records information about the text of each SQL statement, including how many times the database parsed the statement and how

many round-trips the client made to fetch batches of rows while retrieving the query result.

Note: You might need a DBA to grant permission to the FOD account to perform the ALTER SESSION command to do the tracing of SQL output.

Once you've produced a trace file, you can use the TKPROF utility that comes with the database to format the file:

```
tkprof xe_ora_3916.trc trace.prf
```

For details about working with the TKPROF utility, see sections "Understanding SQL Trace and TKPROF" and "Using the SQL Trace Facility and TKPROF" in the *Oracle Database Performance Tuning Guide*.

This will produce a `trace.prf` file containing the interesting information shown in [Example 35-12](#) about the SQL statement performed by the `ClosedOrders` view object. You can see that after initially querying six rows of data in a single execute and fetch from the database, the two subsequent sorts of those results did not cause any further executions. Since the code set the SQL mode to `ViewObject.QUERY_MODE_SCAN_VIEW_ROWS` the `setSortBy()` followed by the `executeQuery()` performed the sort in memory.

Example 35-12 TKPROF Output of a Trace File Confirming Sort Was Done In Memory

```
*****
SELECT * FROM (select o.order_id,
case
when length(o.giftwrap_message) > 5 then
  rtrim(substr(o.giftwrap_message,1,5))||'...'|
else o.giftwrap_message
end as giftwrap_message,
ceil(
  (select trunc(max(creation_date))
   from order_histories
   where order_id = or.order_id)
 - trunc(o.order_date)
) as days_open,
p.email as customer
from orders o, persons p
where o.customer_id = p.person_id
and order status code = 'COMPLETE')

call      count      cpu  elapsed disk  query  current      rows
-----  -----  -----  -----  -----  -----  -----  -----
Parse       1      0.00      0.00    0      0      0          0
Execute     1      0.00      0.00    0      0      0          0
Fetch       1      0.00      0.00    0     22      0          6
-----  -----  -----  -----  -----  -----  -----  -----
total      3      0.00      0.00    0     22      0          6
*****
```

35.6.2.2 Extensibility Points for In-Memory Sorting

Should you need to customize the way that rows are sorted in memory, you have the following two extensibility points:

1. You can override the method:

```
public void sortRows(Row[] rows)
```

This method performs the actual in-memory sorting of rows. By overriding this method you can plug in an alternative sorting approach if needed.

2. You can override the method:

```
public Comparator getRowComparator()
```

The default implementation of this method returns an `oracle.jbo.RowComparator`. `RowComparator` invokes the `compareTo()` method to compare two data values. These methods/objects can be overridden to provide custom compare routines.

35.6.3 Performing In-Memory Filtering with View Criteria

To filter the contents of a row set using `ViewCriteria`, you can call:

- `applyViewCriteria()` or `setApplyViewCriteriaNames()` followed by `executeQuery()` to produce a new, filtered row set.
- `findByViewCriteria()` to retrieve a new row set to process programmatically without changing the contents of the original row set.

Both of these approaches can be used against the database or to perform in-memory filtering, or both, depending on the view criteria mode. You set the criteria mode using the `setCriteriaMode()` method on the `ViewCriteria` object, to which you can pass either of the following integer flags, or the logical OR of both:

- `ViewCriteria.CRITERIA_MODE_QUERY`
- `ViewCriteria.CRITERIA_MODE_CACHE`

When used for in-memory filtering with view criteria, the operators supported are shown in [Table 35-1](#). You can group subexpressions with parenthesis and use the AND and OR operators between subexpressions.

Table 35-1 SQL Operators Supported By In-Memory Filtering with View Criteria

Operator	Operation
=, >, <, <=, >=, <>, LIKE, BETWEEN	.Comparison
NOT	Logical negation
AND	Conjunction
OR	Disjunction

[Example 35-13](#) shows the interesting lines from a `TestClientFindByViewCriteria` class that uses the two features described above both against the database and in-memory. It uses a `CustomerList` view object instance and performs the following basic steps:

1. Queries customers from the database with a last name starting with a 'C', producing the output:

```
--- Initial database results with applied view criteria ---
John Chen
Emerson Clabe
Karen Colmenares
```

2. Subsets the results from step 1 in memory to only those with a first name starting with 'J'. It does this by adding a second view criteria row to the view criteria and setting the conjunction to use "AND". This produces the output:

```
--- After augmenting view criteria and applying in-memory ---  
John Chen
```

3. Sets the conjunction back to OR and re-applies the criteria to the database to query customers with last name like 'J%' or first name like 'C%'. This produces the output:

```
--- After changing view criteria and applying to database again ---  
John Chen  
Jose Manuel Urman  
Emerson Clabe  
Karen Colmenares  
Jennifer Whalen
```

4. Defines a new criteria to find customers in-memory with first or last name that contain a letter 'o'
5. Uses `findByViewCriteria()` to produce new row set instead of subsetting, producing the output:

```
--- Rows returned from in-memory findByViewCriteria ---  
John Chen  
Jose Manuel Urman  
Emerson Clabe  
Karen Colmenares
```

6. Shows that original row set hasn't changed when `findByViewCriteria()` was used, producing the output:

```
--- Note findByViewCriteria didn't change rows in the view ---  
John Chen  
Jose Manuel Urman  
Emerson Clabe  
Karen Colmenares  
Jennifer Whalen
```

Example 35–13 Performing Database and In-Memory Filtering with View Criteria

```
// In TestClientFindByViewCriteria.java  
ViewObject vo = am.findViewObject("CustomerList");  
// 1. Show customers with a last name starting with a 'M'  
ViewCriteria vc = vo.createViewCriteria();  
ViewCriteriaRow vcr1 = vc.createViewCriteriaRow();  
vcr1.setAttribute("LastName", "LIKE 'M%'");  
vo.applyViewCriteria(vc);  
vo.executeQuery();  
vc.add(vcr1);  
vo.executeQuery();  
showRows(vo, "Initial database results with applied view criteria");  
// 2. Subset results in memory to those with first name starting with 'S'  
vo.setQueryMode(ViewObject.QUERY_MODE_SCAN_VIEW_ROWS);  
ViewCriteriaRow vcr2 = vc.createViewCriteriaRow();  
vcr2.setAttribute("FirstName", "LIKE 'S%'");  
vcr2.setConjunction(ViewCriteriaRow.VCROW_CONJ_AND);  
vc.setCriteriaMode(ViewCriteria.CRITERIA_MODE_CACHE);  
vc.add(vcr2);  
vo.executeQuery();  
showRows(vo, "After augmenting view criteria and applying in-memory");
```

```

// 3. Set conjunction back to OR and re-apply to database query to find
// customers with last name like 'H%' or first name like 'S%'
vc.setCriteriaMode(ViewCriteria.CRITERIA_MODE_QUERY);
vo.setQueryMode(ViewObject.QUERY_MODE_SCAN_DATABASE_TABLES);
vcr2.setConjunction(ViewCriteriaRow.VCROW_CONJ_OR);
vo.executeQuery();
showRows(vo, "After changing view criteria and applying to database again");
// 4. Define new criteria to find customers with first or last name like '%o%'
ViewCriteria nameContains0 = vo.createViewCriteria();
ViewCriteriaRow lastContains0 = nameContains0.createViewCriteriaRow();
lastContains0.setAttribute("LastName", "LIKE '%o%'");
ViewCriteriaRow firstContains0 = nameContains0.createViewCriteriaRow();
firstContains0.setAttribute("FirstName", "LIKE '%o%'");
nameContains0.add(firstContains0);
nameContains0.add(lastContains0);
// 5. Use findByViewCriteria() to produce new rowset instead of subsetting
nameContains0.setCriteriaMode(ViewCriteria.CRITERIA_MODE_CACHE);
RowSet rs = (RowSet)vo.findByViewCriteria(nameContains0,
                                         -1, ViewObject.QUERY_MODE_SCAN_VIEW_ROWS);
showRows(rs, "Rows returned from in-memory findByViewCriteria");
// 6. Show that original rowset hasn't changed
showRows(vo, "Note findByViewCriteria didn't change rows in the view");

```

35.6.4 Performing In-Memory Filtering with RowMatch

The RowMatch object provides an even more convenient way to express in-memory filtering conditions. You create a RowMatch object by passing a query predicate expression to the constructor like this:

```

RowMatch rm =
new RowMatch("LastName = 'Popp' or (FirstName like 'A%' and LastName like 'K%')");
;
```

As you do with the SortBy clause, you phrase the RowMatch expression in terms of the view object attribute names, using the supported operators shown in [Table 35–2](#). You can group subexpressions with parenthesis and use the and and or operators between subexpressions.

Table 35–2 SQL Operators Supported By In-Memory Filtering with RowMatch

Operator	Operation
=, >, <, <=, >=, <>, LIKE, BETWEEN	.Comparison
NOT	Logical negation
AND	Conjunction
OR	Disjunction

You can also use a limited set of SQL functions in the RowMatch expression, as shown in [Table 35–3](#).

Table 35–3 SQL Functions Supported By In-Memory Filtering with RowMatch

Operator	Operation
UPPER	Converts all letters in a string to uppercase.
TO_CHAR	Converts a number or date to a string.

Table 35–3 (Cont.) SQL Functions Supported By In-Memory Filtering with RowMatch

Operator	Operation
TO_DATE	Converts a character string to a date format.
TO_TIMESTAMP	Converts a string to timestamp.

Note: While SQL query predicates treat column names in a case-insensitive way, the attribute names in a RowMatch expression are case-sensitive.

35.6.4.1 Applying a RowMatch to a View Object

To apply a RowMatch to your view object, call the `setRowMatch()` method. In contrast to a `ViewCriteria`, the RowMatch is only used for *in-memory* filtering, so there is no "match mode" to set. You can use a RowMatch on view objects in any supported SQL mode, and you will see the results of applying it the next time you call the `executeQuery()` method.

When you apply a RowMatch to a view object, the RowMatch expression can reference the view object's named bind variables using the same `:VarName` notation that you would use in a SQL statement. For example, if a view object had a named bind variable named `StatusCode`, you could apply a RowMatch to it with an expression like:

```
Status = :StatusCode or :StatusCode = '%'
```

[Example 35–14](#) shows the interesting lines of a `TestClientRowMatch` class that illustrate the RowMatch in action. The `CustomerList` view object used in the example has a transient Boolean attribute named `Selected`. The code performs the following basic steps:

1. Queries the full customer list, producing the output:

```
--- Initial database results ---
Neena Kochhar [null]
Lex De Haan [null]
Nancy Greenberg [null]
:

```

2. Marks odd-numbered rows selected by setting the `Selected` attribute of odd rows to `Boolean.TRUE`, producing the output:

```
--- After marking odd rows selected ---
Neena Kochhar [null]
Lex De Haan [true]
Nancy Greenberg [null]
Daniel Faviet [true]
John Chen [null]
Ismael Sciarra [true]
:

```

3. Uses a RowMatch to subset the row set to contain only the select rows, that is, those with `Selected = true`. This produces the output:

```
--- After in-memory filtering on only selected rows ---
Lex De Haan [true]
Daniel Faviet [true]
```

```

Ismael Sciarra [true]
Luis Popp [true]
:

```

4. Further subsets the row set using a more complicated RowMatch expression, producing the output:

```

--- After in-memory filtering with more complex expression ---
Lex De Haan [true]
Luis Popp [true]

```

Example 35–14 Performing In-Memory Filtering with RowMatch

```

// In TestClientRowMatch.java
// 1. Query the full customer list
ViewObject vo = am.findViewObject("CustomerList");
vo.executeQuery();
showRows(vo, "Initial database results");
// 2. Mark odd-numbered rows selected by setting Selected = Boolean.TRUE
markOddRowsAsSelected(vo);
showRows(vo, "After marking odd rows selected");
// 3. Use a RowMatch to subset row set to only those with Selected = true
RowMatch rm = new RowMatch("Selected = true");
vo.setRowMatch(rm);
// Note: Only need to set SQL mode when not defined at design time
vo.setQueryMode(ViewObject.QUERY_MODE_SCAN_VIEW_ROWS);
vo.executeQuery();
showRows(vo, "After in-memory filtering on only selected rows");
// 5. Further subset rowset using more complicated RowMatch expression
rm = new RowMatch("LastName = 'Popp' "+
    "or (FirstName like 'A%' and LastName like 'K%')");
vo.setRowMatch(rm);
vo.executeQuery();
showRows(vo, "After in-memory filtering with more complex expression");
// 5. Remove RowMatch, set query mode back to database, requery to see full list
vo.setRowMatch(null);
vo.setQueryMode(ViewObject.QUERY_MODE_SCAN_DATABASE_TABLES);
vo.executeQuery();
showRows(vo, "After re-querying to see a full list again");

```

35.6.4.2 Using RowMatch to Test an Individual Row

In addition to using a RowMatch to filter a row set, you can also use its `rowQualifies()` method to test whether any individual row matches the criteria it encapsulates. For example:

```

RowMatch rowMatch = new RowMatch("CountryId = 'US'");
if (rowMatch.rowQualifies(row)) {
    System.out.println("Customer is from the United States ");
}

```

35.6.4.3 How a RowMatch Affects Rows Fetched from the Database

Once you apply a RowMatch, if the view object's SQL mode is set to retrieve rows from the database, when you call `executeQuery()` the RowMatch is applied to rows as they are fetched. If a fetched row does not qualify, it is not added to the rowset.

Unlike a SQL WHERE clause, a RowMatch can evaluate expressions involving transient view object attributes and not-yet-posted attribute values. This can be useful to filter queried rows based on RowMatch expressions involving transient view row attributes whose values are calculated in Java. This interesting aspect should be used with care,

however, if your application needs to process a large rowset. Oracle recommends using database-level filtering to retrieve the smallest-possible rowset first, and then using RowMatch as appropriate to subset that list in memory.

35.7 Using View Objects to Work with Multiple Row Types

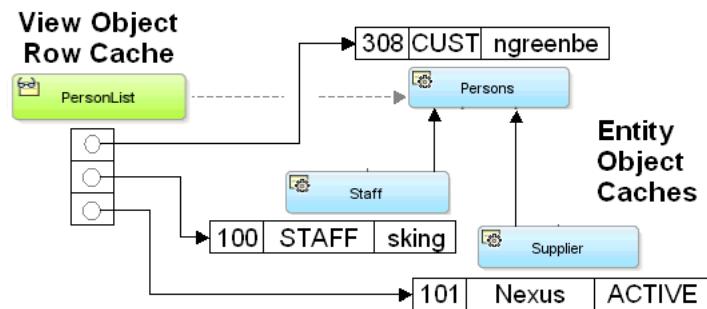
In [Section 34.7, "Using Inheritance in Your Business Domain Layer"](#) you saw how to create an inheritance hierarchy of Persons, Supplier, and Staff entity objects. Sometimes you will create a view object to work with entity rows of a single type like Technician, which perhaps includes Technician-specific attributes. At other times you may want to query and update rows for persons, suppliers, and staff in the same row set, working with attributes that they all share in common.

Note: To experiment with the example described in this section, use the same `InheritanceAndPolymorphicQueries` project in the `AdvancedEntityExamples` workspace used in [Section 34.7, "Using Inheritance in Your Business Domain Layer"](#).

35.7.1 Working with Polymorphic Entity Usages

A *polymorphic* entity usage is one that references a base entity object in an inheritance hierarchy and is configured to handle *subtypes* of that entity as well. [Figure 35–11](#) shows the results of using a view object with a polymorphic entity usage. The entity-based PersonList view object has the Person entity object as its primary entity usage. The view object partitions each row retrieved from the database into an entity row part of the appropriate entity object *subtype* of Person. It creates the appropriate entity row subtype based on consulting the value of the discriminator attribute. For example, if the PersonList query retrieves one row for person `ngreenbe`, one row for staff `sking`, and one row for supplier `ahunold`, the underlying entity row parts would be as shown in the figure.

Figure 35–11 View Object with a Polymorphic Entity Usage Handles Entity Subtypes



35.7.2 How To Create a View Object with a Polymorphic Entity Usage

To create a view object with a polymorphic entity usage, follow these steps:

1. Identify the entity object that represents the base type in the entity inheritance hierarchy you want to work with.
2. Create an entity-based view object with that base entity as its entity usage.

Note: When an entity-based view object references an entity object with a discriminator attribute, then JDeveloper enforces that the discriminator attribute is included in the query as well (in addition to the primary key attribute).

3. On the **Entity Objects** panel of the Create View Object wizard, select the entity usage in the **Selected** list and click **Subtypes...**
- In the **Subtypes** dialog that appears, shuttle the desired entity subtypes you want to allow from the **Available** to the **Selected** list, and click **OK**

Then click **OK** to create the view object.

35.7.3 What Happens When You Create a View Object with a Polymorphic Entity Usage

When you create an entity-based view object with a polymorphic entity usage, JDeveloper adds information about the allowed entity subtypes to the view object's XML component definition. For example, when creating the `PersonList` view object above, the names of the allowed subtype entity objects are recorded in an `<AttrArray>` tag like this:

```
<ViewObject Name="PersonList" ... >
  <EntityUsage Name="ThePerson"
    Entity="devguide.advanced.inheritance.Persons" >
    </EntityUsage>
  ...
  <AttrArray Name="EntityImports">
    <Item Value="devguide.advanced.inheritance.Staff" />
    <Item Value="devguide.advanced.inheritance.Supplier" />
  </AttrArray>
  <!-- etc. -->
</ViewObject>
```

35.7.4 What You May Need to Know

35.7.4.1 Your Query Must Limit Rows to Expected Entity Subtypes

If your view object expects to work with only a *subset* of the available entity subtypes in a hierarchy, you need to include an appropriate WHERE clause that limits the query to only return rows whose discriminator column matches the expected entity types.

35.7.4.2 Exposing Selected Entity Methods in View Rows Using Delegation

By design, clients do not work directly with entity objects. Instead, they work indirectly with entity objects through the view rows of an appropriate view object that presents a relevant set of information related to the task at hand. Just as a view object can expose a particular set of the underlying *attributes* of one or more entity objects related to the task at hand, it can also expose a selected set of *methods* from those entities. You accomplish this by enabling a custom view row Java class and writing a method in the view row class that:

- Accesses the appropriate underlying entity row using the generated entity accessor in the view row, and
- Invokes a method on it

For example, assume that the Persons entity object contains a `performPersonFeature()` method in its `PersonsImpl` class. To expose this method to clients on the `PersonsList` view row, you can enable a custom view row Java class and write the method shown in [Example 35–15](#). JDeveloper generates an entity accessor method in the view row class for each participating entity usage based on the entity usage *alias* name. Since the alias for the Persons entity in the `PersonsList` view object is "ThePerson", it generates a `getThePerson()` method to return the entity row part related to that entity usage.

Example 35–15 Exposing Selected Entity Object Methods on View Rows Through Delegation

```
// In PersonListRowImpl.java
public void performPersonFeature() {
    getThePerson().performPersonFeature();
}
```

The code in the view row's `performPersonFeature()` method uses this `getThePerson()` method to access the underlying `PersonImpl` entity row class and then invokes *its* `performPersonFeature()` method. This style of coding is known as *delegation*, where a view row method delegates the implementation of one of *its* methods to a corresponding method on an underlying entity object. When delegation is used in a view row with a polymorphic entity usage, the delegated method call is handled by appropriate underlying entity row subtype. This means that if the `PersonsImpl`, `StaffImpl`, and `SupplierImpl` classes implement the `performPersonFeature()` method in a different way, the appropriate implementation is used depending on the entity subtype for the current row.

After exposing this method on the client row interface, client programs can use the custom row interface to invoke custom business functionality on a particular view row. [Example 35–16](#) shows the interesting lines of code from a `TestEntityPolymorphism` class. It iterates over all the rows in the `PersonList` view object instance, casts each one to the custom `PersonListRow` interface, and invokes the `performPersonFeature()` method.

Example 35–16 Invoking a View Row Method That Delegates to an Entity Object

```
PersonList personlist = (PersonList)am.findViewObject("PersonList");
personlist.executeQuery();
while (personlist.hasNext()) {
    PersonListRow person = (PersonListRow)personlist.next();
    System.out.print(person.getEmail()+"->");
    person.performPersonFeature();
}
```

Running the client code in [Example 35–16](#) produces the following output:

```
austin->## performPersonFeature as Supplier
hbaer->## performPersonFeature as Person
:
sking->## performPersonFeature as Staff
:
```

Rows related to Persons entities display a message confirming that the `performPersonFeature()` method in the `PersonsImpl` class was used. Rows related to Supplier and Staff entities display a different message, highlighting the different implementations that the respective `SupplierImpl` and `StaffImpl` classes have for the inherited `performPersonFeature()` method.

35.7.4.3 Creating New Rows With the Desired Entity Subtype

In a view object with a polymorphic entity usage, when you create a new view row it contains a new entity row part whose type matches the base entity usage. To create a new view row with one of the entity *subtypes* instead, use the `createAndInitRow()` method. [Example 35–17](#) shows two custom methods in the `PersonList` view object's Java class that use `createAndInitRow()` to allow a client to create new rows having entity rows either of `Staff` or `Supplier` subtypes. To use the `createAndInitRow()`, as shown in the example, create an instance of the `NameValuePairs` object and set it to have an appropriate value for the discriminator attribute. Then, pass that `NameValuePairs` to the `createAndInitRow()` method to create a new view row with the appropriate entity row subtype, based on the value of the discriminator attribute you passed in.

Example 35–17 Exposing Custom Methods to Create New Rows with Entity Subtypes

```
// In PersonListImpl.java
public PersonListRow createStaffRow() {
    NameValuePairs nvp = new NameValuePairs();
    nvp.setAttribute("PersonTypeCode", "STAFF");
    return (PersonListRow)createAndInitRow(nvp);
}
public PersonListRow createSupplierRow() {
    NameValuePairs nvp = new NameValuePairs();
    nvp.setAttribute("PersonTypeCode", "SUPP");
    return (PersonListRow)createAndInitRow(nvp);
}
```

If you expose methods like this on the view object's custom interface, then at runtime, a client can call them to create new view rows with appropriate entity subtypes.

[Example 35–18](#) shows the interesting lines relevant to this functionality from a `TestEntityPolymorphism` class. First, it uses the `createRow()`, `createStaffRow()`, and `createSupplierRow()` methods to create three new view rows. Then, it invokes the `performPersonFeature()` method from the `PersonListRow` custom interface on each of the new rows.

As expected, each row handles the method in a way that is specific to the subtype of entity row related to it, producing the results:

```
## performPersonFeature as Person
## performPersonFeature as Staff
## performPersonFeature as Supplier
```

Example 35–18 Creating New View Rows with Different Entity Subtypes

```
// In TestEntityPolymorphism.java
PersonListRow newPerson = (PersonListRow)Personlist.createRow();
PersonListRow newMgr = Personlist.createStaffRow();
PersonListRow newTech = Personlist.createSupplierRow();
newPerson.performPersonFeature();
newStaff.performPersonFeature();
newSupplier.performPersonFeature();
```

35.7.5 Working with Polymorphic View Rows

In the example shown in [Section 35.7, "Using View Objects to Work with Multiple Row Types"](#), the polymorphism occurs "behind the scenes" at the entity object level. Since the client code works with all view rows using the same `PersonListRow` interface, it cannot distinguish between rows based on a `Staff` entity object from those based on a

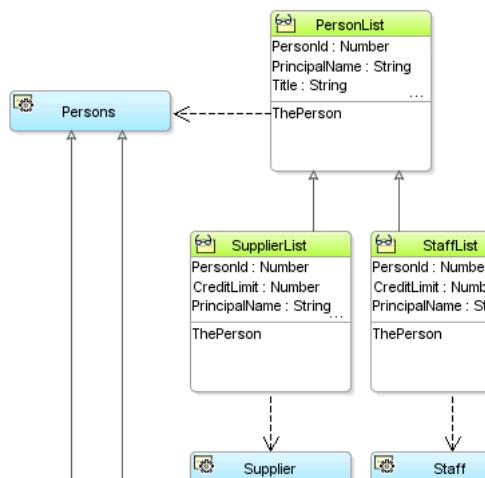
Persons entity object. The code works with all view rows using the same set of view row attributes and methods common to all types of underlying entity subtypes.

If you configure a view object to support polymorphic view rows, then the client can work with different types of view rows using a view row interface specific to the type of row it is. By doing this, the client can access view attributes or invoke view row methods that are specific to a given subtype as needed. [Figure 35–12](#) illustrates the hierarchy of view objects that enables this feature for the PersonList example considered above. SupplierList and StaffList are view objects that extend the base PersonList view object. Notice that each one includes an additional attribute specific to the subtype of Person they have as their entity usage. SupplierList includes an additional ContractExpires attribute, while StaffList includes the additional DiscountEligible attribute. When configured for view row polymorphism as described in the next section, a client can work with the results of the PersonList view object using:

- PersonListRow interface for view rows related to persons
- SupplierListRow interface for view rows related to suppliers
- StaffListRow interface for view rows related to staff

As you'll see, this allows the client to access the additional attributes and view row methods that are specific to a given subtype of view row.

Figure 35–12 Hierarchy of View Object Subtypes Enables View Row Polymorphism



35.7.6 How to Create a View Object with Polymorphic View Rows

To create a view object with polymorphic view rows, follow these steps:

1. Identify an existing view object to be the base
In the example above, the PersonList view object is the base.
2. Identify a discriminator attribute for the view row, and give it a default value.

Open the Attribute Editor and check the **Discriminator** checkbox to mark the attribute as the one that distinguishes which view row interface to use. To view the Attribute Editor, select the attribute and click the **Edit** icon on the Attributes page of the view object's overview editor. You also must supply a value for the **Value** field that matches the attribute value for which you expect the base view object's view row interface to be used. For example, in the PersonList view object, you

would mark the PersonTypeCode attribute as the discriminator attribute and supply a default value of "person".

3. Enable a custom view row class for the base view object, and expose at least one method on the client row interface. This can be one or all of the view row attribute accessor methods, as well as any custom view row methods.

4. Create a new view object that *extends* the base view object

In the example above, SupplierList extends the base PersonList view object.

5. Enable a custom view row class for the extended view object.

If appropriate, add additional custom view row methods or override custom view row methods inherited from the parent view object's row class.

6. Supply a *distinct* value for the discriminator attribute in the extended view object.

The SupplierList view object provides the value of "SUPP" for the PersonTypeCode discriminator attribute.

7. Repeat steps 4-6 to add additional extended view objects as needed.

For example, the StaffList view object is a second one that extends PersonList. It supplies the value "STAFF" for the PersonTypeCode discriminator attribute.

After setting up the view object hierarchy, you need to define the list of view object subtypes that participate in the view row polymorphism. To accomplish this, do the following:

1. Add an instance of each type of view object in the hierarchy to the data model of an application module.

For example, the PersonModule application module in the example has instances of PersonList, SupplierList, and StaffList view objects.

2. In the **Data Model** panel of the Application Module Editor, click **Subtypes...**
3. In the **Subtypes** dialog that appears, shuttle the desired view object subtypes that you want to participate in view row polymorphism from the **Available** to the **Selected** list, and click **OK**

35.7.7 What You May Need to Know

35.7.7.1 Selecting Subtype-Specific Attributes in Extended View Objects

When you create an extended view object, it inherits the entity usage of its parent. If the parent view object's entity usage is based on an entity object with subtypes in your domain layer, you may want your extended view object to work with one of these subtypes instead of the inherited parent entity usage type. Two reasons you might want to do this are:

1. To select attributes that are *specific* to the entity subtype
2. To be able to write view row methods that delegate to methods *specific* to the entity subtype

In order to do this, you need to override the inherited entity usage to refer to the desired entity subtype. To do this, perform these steps in the View Object Editor for your extended view object:

1. On the **Entity Objects** panel, verify that you are working with an extended entity usage.

For example, when creating the `SupplierList` view object that extends the `PersonList` view object, the entity usage with the alias `ThePerson` will initially display in the **Selected** list as: **ThePerson(Person): extended**. The type of the entity usage is in parenthesis, and the "extended" label confirms that the entity usage is currently inherited from its parent.

2. Select the desired entity *subtype* in the **Available** list that you want to override the inherited one. It must be a subtype entity of the existing entity usage's type.

For example, you would select the `Supplier` entity object in the **Available** list to overridden the inherited entity usage based on the `Persons` entity type.

3. Click **>** to shuttle it to the **Selected** list
4. Acknowledge the alert that appears, confirming that you want to override the existing, inherited entity usage.

When you have performed these steps, the **Selected** list updates to reflect the overridden entity usage. For example, for the `SupplierList` view object, after overriding the `Persons`-based entity usage with the `Supplier` entity subtype, it updates to show: **ThePerson (Supplier): overridden**.

After overriding the entity usage to be related to an entity subtype, you can then use the **Attributes** tab of the editor to select additional attributes that are specific to the subtype. For example, the `SupplierList` view object includes the additional attribute named `ContractExpires` that is specific to the `Supplier` entity object.

35.7.7.2 Delegating to Subtype-Specific Methods After Overriding the Entity Usage

After overriding the entity usage in an extended view object to reference a subtype entity, you can write view row methods that delegate to methods specific to the subtype entity class. [Example 35–19](#) shows the code for a

`performSupplierFeature()` method in the custom view row class for the `SupplierList` view object. It casts the return value from the `getThePerson()` entity row accessor to the subtype `SupplierImpl`, and then invokes the `performSupplierFeature()` method that is specific to `Supplier` entity objects.

Example 35–19 View Row Method Delegating to Method in Subtype Entity

```
// In SupplierListRowImpl.java
public void performSupplierFeature() {
    SupplierImpl supplier = (SupplierImpl) getThePerson();
    supplier.performSupplierFeature();
}
```

Note: You need to perform the explicit cast to the entity subtype here because JDeveloper does not yet take advantage of the JDK feature called covariant return types that would allow a subclass like `SupplierListRowImpl` to override a method like `getThePerson()` and change its return type.

35.7.7.3 Working with Different View Row Interface Types in Client Code

[Example 35–20](#) shows the interesting lines of code from a `TestViewRowPolymorphism` class that performs the following steps:

1. Iterates over the rows in the PersonList view object.
For each row in the loop, it uses Java's instanceof operator to test whether the current row is an instance of the StaffListRow or the SupplierListRow.
2. If the row is a StaffListRow, then cast it to this more specific type and:
 - Call the performStaffFeature() method specific to the StaffListRow interface, and
 - Access the value of the DiscountEligible attribute that is specific to the StaffList view object.
3. If the row is a SupplierListRow, then cast it to this more specific type and:
 - Call the performSupplierFeature() method specific to the SupplierListRow interface, and
 - Access the value of the ContractExpires attribute that is specific to the SupplierList view object.
4. Otherwise, just call a method on the PersonListRow

Example 35–20 Using View Row Polymorphism in Client Code

```
// In TestViewRowPolymorphism.java
ViewObject vo = am.findViewObject("PersonList");
vo.executeQuery();
// 1. Iterate over the rows in the PersonList view object
while (vo.hasNext()) {
    PersonListRow Person = (PersonListRow)vo.next();
    System.out.print(Person.getEmail()+"->");
    if (Person instanceof StaffListRow) {
        // 2. If the row is a StaffListRow, cast it
        StaffListRow mgr = (StaffListRow)Person;
        mgr.performStaffFeature();
        System.out.println("Discount Status: "+staff.getDiscountEligible());
    }
    else if (Person instanceof SupplierListRow) {
        // 3. If the row is a StaffListRow, cast it
        SupplierListRow tech = (SupplierListRow)Person;
        supplier.performSupplierFeature();
        System.out.println("Contract expires: "+tech.getContractExpires());
    }
    else {
        // 4. Otherwise, just call a method on the PersonListRow
        Person.performPersonFeature();
    }
}
```

Running the code in Example 35–20 produces the following output:

```
daustin->## performSupplierFeature called
Contract expires: 2006-05-09
hbaer->## performPersonFeature as Person
:
sking->## performStaffFeature called
Discount Status: Y
:
```

This illustrates that by using the view row polymorphism feature the client was able to distinguish between view rows of different types and access methods and attributes specific to each subtype of view row.

35.7.7.4 View Row Polymorphism and Polymorphic Entity Usage are Orthogonal

While often even more useful when used *together*, the view row polymorphism and the polymorphic entity usage features are distinct and can be used separately. In particular, the view row polymorphism feature can be used for read-only view objects, as well as for entity-based view objects. When you combine both mechanisms, you can have both the entity row part being polymorphic, as well as the view row type.

Note to use view row polymorphism with either view objects or entity objects, you must configure the discriminator attribute property separately for each. This is necessary because read-only view objects contain no related entity usages from which to infer the discriminator information.

In summary, to use view row polymorphism:

1. Configure an attribute to be the discriminator at the view object level in the root view object in an inheritance hierarchy.
2. Have a hierarchy of inherited view objects each of which provides a distinct value for the "Default Value" property of that view object level discriminator attribute.
3. List the subclassed view objects in this hierarchy in the application module's list of Subtypes.

Whereas, to create a view object with a polymorphic entity usage:

1. Configure an attribute to be the discriminator at the entity object level in the root entity object in an inheritance hierarchy.
2. Have a hierarchy of inherited entity objects, each of which overrides and provides a distinct value for the "Default Value" property of that entity object level discriminator attribute.
3. List the subclassed entity objects in a view object's list of Subtypes.

35.8 Reading and Writing XML

The Extensible Markup Language (XML) standard from the Worldwide Web Consortium (W3C) defines a language-neutral approach for electronic data exchange. Its rigorous set of rules enables the structure inherent in data to be easily encoded and unambiguously interpreted using human-readable text documents.

View objects support the ability to write these XML documents based on their queried data. View objects also support the ability to read XML documents in order to apply changes to data including inserts, updates, and deletes. When you've introduced view links, this XML capability supports reading and writing multi-level nested information for master/detail hierarchies of any complexity. While the XML produced and consumed by view objects follows a canonical format, you can combine the view object's XML features with XML Stylesheet Language Transformations (XSLT) to easily convert between this canonical XML format and any format you need to work with.

35.8.1 How to Produce XML for Queried Data

To produce XML from a view object, use the `writeXML()` method. It offers two ways to control the XML produced:

1. For precise control over the XML produced, you can specify a view object attribute map indicating which attributes should appear, including which view link accessor attributes should be accessed for nested, detail information:

```
Node writeXML(long options, HashMap voAttrMap)
```

2. To produce XML that includes all attributes, you can simply specify a depth-level that indicates how many levels of view link accessor attributes should be traversed to produce the result:

```
Node writeXML(int depthCount, long options)
```

The options parameter is an integer flag field that can be set to one of the following bit flags:

- `XMLInterface.XML_OPT_ALL_ROWS`
Includes all rows in the view object's row set in the XML.
- `XMLInterface.XML_OPT_LIMIT_RANGE`
Includes only the rows in the current range in the XML.

Using the logical OR operation, you can combine either of the above flags with the `XMLInterface.XML_OPT_ASSOC_CONSISTENT` flag when you want to include new, unposted rows in the current transaction in the XML output.

Both versions of the `writeXML()` method accept an optional third argument which is an XSLT stylesheet that, if supplied, is used to transform the XML output before returning it.

35.8.2 What Happens When You Produce XML

When you produce XML using `writeXML()`, the view object begins by creating a wrapping XML element whose default name matches the name of the view object definition. For example, for a `Persons` view object in the `devguide.advanced.xml.queries` package, the XML produced will be wrapped in an outermost `<Persons>` tag.

Then, it converts the attribute data for the appropriate rows into XML elements. By default, each row's data is wrapped in a row element whose name is the name of the view object with the Row suffix. For example, each row of data from a view object named `Persons` is wrapped in an `<PersonsRow>` element. The elements representing the attribute data for each row appear as nested children inside this row element.

If any of the attributes is a view link accessor attribute, and if the parameters passed to `writeXML()` enable it, the view object will include the data for the detail rowset returned by the view link accessor. This nested data is wrapped by an element whose name is determined by the name of the view link accessor attribute. The return value of the `writeXML()` method is an object that implements the standard W3C Node interface, representing the root element of the generated XML.

Note: The `writeXML()` method uses view link accessor attributes to programmatically access detail collections. It does not require adding view link instances in the data model.

For example, to produce an XML element for all rows of a `Persons` view object instance, and following view link accessors as many levels deep as exists, [Example 35-21](#) shows the code required.

Example 35-21 Generating XML for All Rows of a View Object to All View Link Levels

```
ViewObject vo = am.findViewObject("PersonsView");
printXML(vo.writeXML(-1,XMLInterface.XML_OPT_ALL_ROWS));
```

The Persons view object is linked to a Orders view object showing the service requests created by that person. In turn, the Orders view object is linked to a OrderItems view object providing details on the notes entered for the service request by customers and technicians. Running the code in [Example 35–21](#) produces the XML shown in [Example 35–22](#), reflecting the nested structure defined by the view links.

Example 35–22 XML from a Users View Object with Two Levels of View Linked Details

```
...
<PersonsViewRow>
    <PersonId>111</PersonId>
    <PrincipalName>ISCIARRA</PrincipalName>
    <FirstName>Ismael</FirstName>
    <LastName>Sciarra</LastName>
    <PersonTypeCode>CUST</PersonTypeCode>
    <ProvisionedFlag>N</ProvisionedFlag>
    <PrimaryAddressId>42</PrimaryAddressId>
    <MembershipId>2</MembershipId>
    <Email>ISCIARRA</Email>
    <ConfirmedEmail>ISCIARRA</ConfirmedEmail>
    <PhoneNumber>228.555.0126</PhoneNumber>
    <DateOfBirth>1971-09-30</DateOfBirth>
    <MaritalStatusCode>SING</MaritalStatusCode>
    <Gender>M</Gender>
    <ContactableFlag>Y</ContactableFlag>
    <ContactByAffilliatesFlag>Y</ContactByAffilliatesFlag>
    <CreatedBy>SEED_DATA</CreatedBy>
    <CreationDate>2008-08-15 11:26:36.0</CreationDate>
    <LastUpdatedBy>SEED_DATA</LastUpdatedBy>
    <LastUpdateDate>2008-08-15 11:26:36.0</LastUpdateDate>
    <ObjectVersionId>1</ObjectVersionId>
    <OrdersView>
        <OrdersViewRow>
            <OrderId>1017</OrderId>
            <OrderDate>2008-08-06 11:28:26.0</OrderDate>
            <OrderStatusCode>STOCK</OrderStatusCode>
            <OrderTotal>1649.92</OrderTotal>
            <CustomerId>111</CustomerId>
            <ShipToAddressId>8</ShipToAddressId>
            <ShippingOptionId>2</ShippingOptionId>
            <PaymentOptionId>1006</PaymentOptionId>
            <DiscountId>3</DiscountId>
            <FreeShippingFlag>Y</FreeShippingFlag>
            <CustomerCollectFlag>Y</CustomerCollectFlag>
            <CollectionWarehouseId>102</CollectionWarehouseId>
            <GiftwrapFlag>N</GiftwrapFlag>
            <CreatedBy>0</CreatedBy>
            <CreationDate>2008-08-15 11:28:26.0</CreationDate>
            <LastUpdatedBy>0</LastUpdatedBy>
            <LastUpdateDate>2008-08-15 11:28:26.0</LastUpdateDate>
            <ObjectVersionId>0</ObjectVersionId>
            <OrderItemsView>
                <OrderItemsViewRow>
                    <OrderId>1017</OrderId>
                    <LineItemId>1</LineItemId>
                    <ProductId>22</ProductId>
                    <Quantity>1</Quantity>
                    <UnitPrice>199.95</UnitPrice>
                    <CreatedBy>0</CreatedBy>
                    <CreationDate>2008-08-15 11:32:26.0</CreationDate>
```

```

<LastUpdatedBy>0</LastUpdatedBy>
<LastUpdateDate>2008-08-15 11:32:26.0</LastUpdateDate>
<ObjectVersionId>0</ObjectVersionId>
</OrderItemsViewRow>
<OrderItemsViewRow>
    <OrderId>1017</OrderId>
    <LineItemId>2</LineItemId>
    <ProductId>9</ProductId>
    <Quantity>1</Quantity>
    <UnitPrice>129.99</UnitPrice>
    <CreatedBy>0</CreatedBy>
    <CreationDate>2008-08-15 11:32:27.0</CreationDate>
    <LastUpdatedBy>0</LastUpdatedBy>
    <LastUpdateDate>2008-08-15 11:32:27.0</LastUpdateDate>
    <ObjectVersionId>0</ObjectVersionId>
</OrderItemsViewRow>
<OrderItemsViewRow>
    <OrderId>1017</OrderId>
    <LineItemId>3</LineItemId>
    <ProductId>36</ProductId>
    <Quantity>2</Quantity>
    <UnitPrice>659.99</UnitPrice>
    <CreatedBy>0</CreatedBy>
    <CreationDate>2008-08-15 11:32:27.0</CreationDate>
    <LastUpdatedBy>0</LastUpdatedBy>
    <LastUpdateDate>2008-08-15 11:32:27.0</LastUpdateDate>
    <ObjectVersionId>0</ObjectVersionId>
</OrderItemsViewRow>
</OrderItemsView>
</OrdersViewRow>
</OrdersView>
</PersonsViewRow>
...

```

35.8.3 What You May Need to Know

35.8.3.1 Controlling XML Element Names

You can use the Property Inspector to change the default XML element names used in the view object's canonical XML format by setting several properties. To accomplish this, open the overview editor for the view object, then:

- Select the attribute on the Attributes page and in the Property Inspector, select the Custom Properties navigation tab and set the custom *attribute-level* property named **Xml Element** to a value *SomeOtherName* to change the XML element name used for that attribute to <*SomeOtherName*>

For example, the *Email* attribute in the *Persons* view object defines this property to change the XML element you see in [Example 35-22](#) to be <*EmailAddress*> instead of <*Email*>.

- Select the General navigation tab in the Property Inspector and set the custom *view object-level* property named **Xml Row Element** to a value *SomeOtherRowName* to change the XML element name used for that view object to <*SomeOtherRowName*>.

For example, the Persons view object defines this property to change the XML element name for the rows you see in [Example 35–22](#) to be <Person> instead of <PersonsRow>.

- To change the name of the element names that wrapper nested row set data from view link attribute accessors, use the **View Link Properties** dialog. To open the dialog, in the view link's overview editor, click the **Edit** icon on the Accessors section of Relationship page. Enter the desired name of the view link accessor attribute in the **Accessor Name** field.

35.8.3.2 Controlling Element Suppression for Null-Valued Attributes

By default, if a view row attribute is `null`, then its corresponding element is omitted from the generated XML. Select the attribute on the Attributes page of the overview editor and in the Property Inspector, select the Custom Properties navigation tab and set the custom *attribute*-level property named **Xml Explicit Null** to any value (e.g. "true" or "yes") to cause an element to be included for the attribute if its value is null. For example, if an attribute named `AssignedDate` has this property set, then a row containing a `null` assigned date will contain a corresponding `<AssignedDate null="true"/>` element. If you want this behavior for all attributes of a view object, you can define the **Xml Explicit Null** custom property at the view object level as a shortcut for defining it on each attribute.

35.8.3.3 Printing or Searching the Generated XML Using XPath

Two of the most common things you might want to do with the `XMLNode` object returned from `writeXML()` are:

1. Printing the node to its serialized text representation — to send across the network or save in a file, for example
2. Searching the generated XML using W3C XPath expressions

Unfortunately, the standard W3C Document Object Model (DOM) API does not include methods for doing *either* of these useful operations. But there is hope. Since ADF Business Components uses the Oracle XML parser's implementation of the DOM, you can cast the `Node` return value from `writeXML()` to the Oracle specific classes `XMLNode` or `XMLElement` (in the `oracle.xml.parser.v2` package) to access additional useful functionality like:

- Printing the XML element to its serialized text format using the `print()` method
- Searching the XML element in memory with XPath expressions using the `selectNodes()` method
- Finding the value of an XPath expression related to the XML element using the `valueOf()` method.

[Example 35–23](#) shows the `printXML()` method in the `TestClientWriteXML`. It casts the `Node` parameter to an `XMLNode` and calls the `print()` method to dump the XML to the console.

Example 35–23 Using the `XMLNode`'s `print()` Method to Serialize XML

```
// In TestClientWriteXML.java
private static void printXML(Node n) throws IOException {
    ((XMLNode)n).print(System.out);
}
```

35.8.3.4 Using the Attribute Map For Fine Control Over Generated XML

When you need fine control over which attributes appear in the generated XML, use the version of the `writeXML()` method that accepts a `HashMap`. [Example 35–24](#) shows the interesting lines from a `TestClientWriteXML` class that use this technique. After creating the `HashMap`, you put `String[]`-valued entries into it containing the names of the attributes you want to include in the XML, keyed by the fully-qualified name of the view definition those attributes belong to. The example includes the `PersonId`, `Email`, `PersonTypeCode`, and `OrdersView` attributes from the `Persons` view object, and the `OrderId`, `OrderStatusCode`, and `OrderTotal` attributes from the `OrdersView` view object.

Note: For upward compatibility reasons with earlier versions of ADF Business Components the `HashMap` expected by the `writeXML()` method is the one in the `com.sun.java.util.collections` package.

While processing the view rows for a given view object instance:

- If an entry exists in the attribute map with a key matching the fully-qualified view definition name for that view object, then only the attributes named in the corresponding `String` array are included in the XML.
Furthermore, if the string array includes the name of a view link accessor attribute, then the nested contents of its detail row set are included in the XML. If a view link accessor attribute name does not appear in the string array, then the contents of its detail row set are not included.
- If no such entry exists in the map, then *all* attributes for that row are included in the XML.

Example 35–24 Using a View Definition Attribute Map for Fine Control Over Generated XML

```
HashMap viewDefMap = new HashMap();
viewDefMap.put("devguide.advanced.xml.queries.PersonsView",
    new String[]{"PersonId", "Email", "PersonTypeCode",
        "OrdersView" /* View link accessor attribute */});
viewDefMap.put("devguide.advanced.xml.queries.OrdersView",
    new String[]{"OrderId", "OrderStatusCode", "OrderTotal"});
printXML(vo.writeXML(XMLInterface.XML_OPT_ALL_ROWS, viewDefMap));
```

Running the example produces XML that includes only the exact attributes and view link accessors indicated by the supplied attribute map.

35.8.3.5 Use the Attribute Map Approach with Bi-Directional View Links

If your view objects are related through a view link that you have configured to be bi-directional, then you must use the `writeXML()` approach that uses the attribute map. If you were to use the `writeXML()` approach in the presence of bi-directional view links and were to supply a maximum depth of `-1` to include all levels of view links that exist, the `writeXML()` method will go into an infinite loop as it follows the bi-directional view links back and forth, generating deeply nested XML containing duplicate data until it runs out of memory. Use `writeXML()` with an attribute map instead in this situation. Only by using this approach can you control which view link accessors are included in the XML and which are not to avoid infinite recursion while generating the XML.

35.8.3.6 Transforming Generated XML Using an XSLT Stylesheet

When the canonical XML format produced by `writeXML()` does not meet your needs, you can supply an XSLT stylesheet as an optional argument. It will produce the XML as it would normally, but then transform that result using the supplied stylesheet before returning the final XML to the caller.

Consider the XSLT stylesheet shown in [Example 35–25](#). It is a simple transformation with a single template that matches the root element of the generated XML from [Example 35–25](#) to create a new `<CustomerEmailAddresses>` element in the result. The template uses the `<xsl:for-each>` instruction to process all `<PersonsView>` elements that contain more than one `<OrdersViewRow>` child element inside a nested `<OrdersViews>` element. For each `<PersonsView>` element that qualifies, it creates a `<Customer>` element in the result whose `Contact` attribute is populated from the value of the `<Email>` child element of the `<PersonsView>`.

Example 35–25 XSLT Stylesheet to Transform Generated XML Into Another Format

```
<?xml version="1.0" encoding="windows-1252" ?>
<xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
    <xsl:template match="/">
        <CustomerEmailAddresses>
            <xsl:for-each
                select="/PersonsView/PersonsViewRow[count(OrdersView/OrdersViewRow) >
                    1]">
                <xsl:sort select="Email"/>
                <Customer Contact="{Email}" />
            </xsl:for-each>
        </CustomerEmailAddresses>
    </xsl:template>
</xsl:stylesheet>
```

[Example 35–26](#) shows the interesting lines from a `TestClientWriteXML` class that put this XSLT stylesheet into action when calling `writeXML()`.

Example 35–26 Passing an XSLT Stylesheet to `writeXML()` to Transform the Resulting XML

```
// In TestClientWriteXML.java
XSLStylesheet xsl = getXSLStylesheet();
printXML(vo.writeXML(XMLInterface.XML_OPT_ALL_ROWS,viewDefMap,xsl));
```

Running the code in [Example 35–26](#) produces the transformed XML shown here:

```
<CustomerEmailAddresses>
    <Customer Contact="dfaviet"/>
    <Customer Contact="jchen"/>
    <Customer Contact="ngreenbe"/>
</CustomerEmailAddresses>
```

The `getXSLStylesheet()` helper method shown in [Example 35–27](#) is also interesting to study since it illustrates how to read a resource like an XSLT stylesheet from the classpath at runtime. The code expects the `Example.xsl` stylesheet to be in the same directory as the `TestClientWriteXML` class. By referencing the `Class` object for the `TestClientWriteXML` class using the `.class` operator, the code uses the `getResource()` method to get a URL to the resource. Then, it passes the URL to the `newXSLStylesheet()` method of the `XSLProcessor` class to create a new `XSLStylesheet` object to return. That object represents the compiled version of the XSLT stylesheet read in from the `*.xsl` file.

Example 35–27 Reading an XSLT Stylesheet as a Resource from the Classpath

```
private static XSLStylesheet getXSLStylesheet()
    throws XMLParseException, SAXException, IOException, XSLException {
    String xslurl = "Example.xsl";
    URL xslURL = TestClientWriteXML.class.getResource(xslurl);
    XSLProcessor xslProc = new XSLProcessor();
    return xslProc.newXSLStylesheet(xslURL);
}
```

Note: When working with resources like XSLT stylesheets that you want to be included in the output directory along with your compiled Java classes and XML metadata, you can use the **Compiler** page of the **Project Properties** dialog to update the **Copy File Types to Output Directory** field to include `.xsl` in the semicolon-separated list.

35.8.3.7 Generating XML for a Single Row

In addition to calling `writeXML()` on a view object, you can call the same method with the same parameters and options on any `Row` as well. If the `Row` object on which you call `writeXML()` is a entity row, you can bitwise-OR the additional `XMLInterface.XML_OPT_CHANGES_ONLY` flag if you only want the changed entity attributes to appear in the XML.

35.8.4 How to Consume XML Documents to Apply Changes

To have a view object consume an XML document to process inserts, updates, and deletes, use the `readXML()` method:

```
void readXML(Element elem, int depthcount)
```

The canonical format expected by `readXML()` is the same as what would be produced by a call to the `writeXML()` method on the same view object. If the XML document to process does not correspond to this canonical format, you can supply an XSLT stylesheet as an optional third argument to `readXML()` to transform the incoming XML document *into* the canonical format before it is read for processing.

35.8.5 What Happens When You Consume XML Documents

When a view object consumes an XML document in canonical format, it processes the document to recognize row elements, their attribute element children, and any nested elements representing view link accessor attributes. It processes the document recursively to a maximum level indicated by the `depthcount` parameter. Passing `-1` for the `depthcount` to request that it process all levels of the XML document.

35.8.5.1 How `ViewObject.readXML()` Processes an XML Document

For each row element it recognizes, the `readXML()` method does the following:

- Identifies the related view object to process the row.
- Reads the children attribute elements to get the values of the primary key attributes for the row.
- Performs a `findByPrimaryKey()` using the primary key attributes to detect whether the row already exists or not.
- If the row exists:

- If the row element contains the marker attribute `bc4j-action="remove"`, then the existing row is deleted.
- Otherwise, the row's attributes are updated using the values in any attribute element children of the current row element in the XML
- If the row does not exist, then a new row is created, inserted into the view object's rowset. Its attributes are populated using the values in any attribute element children of the current row element in the XML.

35.8.5.2 Using `readXML()` to Processes XML for a Single Row

The same `readXML()` method is also supported on any Row object. The canonical XML format it expects is the same format produced by a call to `writeXML()` on the same row. You can invoke `readXML()` method on a row to:

- Update its attribute values from XML
- Remove the row, if the `bc4j-action="remove"` marker attribute is present on the corresponding row element.
- Insert, update, or delete any nested rows via view link accessors

Consider the XML document shown in [Example 35–28](#). It is in the canonical format expected by a single row in the Technicians view object. Nested inside the root `<TechniciansRow>` element, the `<City>` attribute represents the technician's city. The nested `<ExpertiseAreas>` element corresponds to the `ExpertiseAreas` view link accessor attribute and contains three `<ExpertiseAreasRow>` elements. Each of these includes `<ProdId>` elements representing part of the two-attribute primary key of a `ExpertiseAreas` row. The other primary key attribute, `UserId`, is inferred from the enclosing parent `<TechniciansRow>` element.

Example 35–28 XML Document in Canonical Format to Insert, Update, and Delete Rows

```
<TechniciansRow>
    <!-- This will update Technician's City attribute -->
    <City>Padova</City>
    <ExpertiseAreas>
        <!-- This will be an update since it does exist -->
        <ExpertiseAreasRow>
            <ProdId>100</ProdId>
            <ExpertiseLevel>Expert</ExpertiseLevel>
        </ExpertiseAreasRow>
        <!-- This will be an insert since it doesn't exist -->
        <ExpertiseAreasRow>
            <ProdId>110</ProdId>
            <ExpertiseLevel>Expert</ExpertiseLevel>
        </ExpertiseAreasRow>
        <!-- This will be deleted -->
        <ExpertiseAreasRow bc4j-action="remove">
            <ProdId>112</ProdId>
        </ExpertiseAreasRow>
    </ExpertiseAreas>
</TechniciansRow>
```

[Example 35–29](#) shows the interesting lines of code from a `TestClientReadXML` class that applies this XML datagram to a particular row in the `Technicians` view object, and to the nested set of the technician's areas of expertise via the view link accessor attribute to the `ExpertiseAreas` view object. `TestClientReadXML` class performs the following basic steps:

1. Finds a target row by key (e.g. for customer "jchen").
2. Shows the XML produced for the row before changes are applied.
3. Obtains the parsed XML document with changes to apply using a helper method.
4. Reads the XML document to apply changes to the row.
5. Shows the XML with the pending changes applied.

The `TestClientReadXML` class is using the `XMLInterface.XML_OPT_ASSOC_CONSISTENT` flag described in [Section 35.8.1, "How to Produce XML for Queried Data"](#) to ensure that new, unposted rows are included in the XML.

Example 35–29 Applying Changes to an Existing Row with `readXML()`

```
ViewObject vo = am.findViewObject("CustomersView");
Key k = new Key(new Object[] { 303 });
// 1. Find a target row by key (e.g. for customer "jchen")
Row jchen = vo.findByKey(k, 1)[0];
// 2. Show the XML produced for the row before changes are applied
printXML(jchen.writeXML(-1, XMLInterface.XML_OPT_ALL_ROWS));
// 3. Obtain parsed XML document with changes to apply using helper method
Element xmlToRead = getInsertUpdateDeleteXMLGram();
printXML(xmlToRead);
// 4. Read the XML document to apply changes to the row
jchen.readXML(getInsertUpdateDeleteXMLGram(), -1);
// 5. Show the XML with the pending changes applied
printXML(jchen.writeXML(-1, XMLInterface.XML_OPT_ALL_ROWS |
                           XMLInterface.XML_OPT_ASSOC_CONSISTENT));
```

Running the code in [Example 35–29](#) initially displays the "before" version of Alexander Hunold's information. Notice that:

- The `City` attribute has the value "Southlake"
- The expertise area for product 100 has a level of "Qualified"
- There is an expertise row for product 112, and
- There is no expertise area row related to product 110.

```
<TechniciansRow>
  <UserId>303</UserId>
  <UserRole>technician</UserRole>
  <Email>ahunold</Email>
  :
  <City>Southlake</City>
  :
  <ExpertiseAreas>
    <ExpertiseAreasRow>
      <ProdId>100</ProdId>
      <UserId>303</UserId>
      <ExpertiseLevel>Qualified</ExpertiseLevel>
    </ExpertiseAreasRow>
    :
    <ExpertiseAreasRow>
      <ProdId>112</ProdId>
      <UserId>303</UserId>
      <ExpertiseLevel>Expert</ExpertiseLevel>
    </ExpertiseAreasRow>
    :
  </ExpertiseAreas>
</TechniciansRow>
```

After applying the changes from the XML document using `readXML()` to the row and printing its XML again using `writeXML()` you see that:

- The City is now "Padova"
- The expertise area for product 100 has a level of "Expert"
- The expertise row for product 112 is removed, and
- A new expertise area row for product 110 got created.

```
<TechniciansRow>
  <UserId>303</UserId>
  <UserRole>technician</UserRole>
  <Email>ahunold</Email>
  :
  <City>Padova</City>
  :
  <ExpertiseAreas>
    <ExpertiseAreasRow>
      <ProdId>110</ProdId>
      <UserId>303</UserId>
      <ExpertiseLevel>Expert</ExpertiseLevel>
    </ExpertiseAreasRow>
    <ExpertiseAreasRow>
      <ProdId>100</ProdId>
      <UserId>303</UserId>
      <ExpertiseLevel>Expert</ExpertiseLevel>
    </ExpertiseAreasRow>
  </TechniciansRow>
```

Note: The example illustrated using `readXML()` to apply changes to a single row. If the XML document contained a wrapping `<CustomersView>` row, including the primary key attribute in each of its one or more nested `<CustomersViewRow>` elements, then that document could be processed using the `readXML()` method on the `CustomersView` view object for handling operations for multiple `CustomersView` rows.

35.9 Using Programmatic View Objects for Alternative Data Sources

By default view objects read their data from the database and automate the task of working with the Java Database Connectivity (JDBC) layer to process the database result sets. However, by overriding appropriate methods in its custom Java class, you can create a view object that programmatically retrieves data from alternative data sources like a REF CURSOR, an in-memory array, or a Java *.properties file, to name a few.

35.9.1 How to Create a Read-Only Programmatic View Object

To create a read-only programmatic view object, use the Create View Object wizard and follow these steps:

1. In the **Name** panel, provide a name and package for the view object. For the data source, select **Rows Populated Programmatically, not Based on a Query**.

2. In the **Attributes** panel, click **New** one or more times to define the view object attributes your programmatic view object requires.
3. In the **Attribute Settings** panel, adjust any setting you may need to for the attributes you defined.
4. In the **Java** panel, enable a custom view object class (`ViewObjImpl`) to contain your code.
5. Click **Finish** to create the view object.

In your view object's custom Java class, override the methods described in [Section 35.9.3, "Key Framework Methods to Override for Programmatic View Objects"](#) to implement your custom data retrieval strategy.

35.9.2 How to Create an Entity-Based Programmatic View Object

To create a entity-based view object with programmatic data retrieval, create the view object in the normal way, enable a custom Java class for it, and override the methods described in the next section to implement your custom data retrieval strategy.

35.9.3 Key Framework Methods to Override for Programmatic View Objects

A programmatic view object typically overrides all of the following methods of the base `ViewObjectImpl` class to implement its custom strategy for retrieving data:

- `create()`

This method is called when the view object instance is created and can be used to initialize any state required by the programmatic view object. At a minimum, this overridden method will contain the following lines to ensure the programmatic view object has no trace of a SQL query related to it:

```
// Wipe out all traces of a query for this VO
getViewDef().setQuery(null);
getViewDef().setSelectClause(null);
setQuery(null);
```

- `executeQueryForCollection()`

This method is called whenever the view object's query needs to be executed (or re-executed).

- `hasNextForCollection()`

This method is called to support the `hasNext()` method on the row set iterator for a row set created from this view object. Your implementation returns `true` if you have not yet exhausted the rows to retrieve from your programmatic data source.

- `createRowFromResultSet()`

This method is called to populate each row of "fetched" data. Your implementation will call `createNewRowForCollection()` to create a new blank row and then `populateAttributeForRow()` to populate each attribute of data for the row.

- `getQueryHitCount()`

This method is called to support the `getEstimatedRowCount()` method. Your implementation returns a count, or estimated count, of the number of rows that will be retrieved by the programmatic view object's query.

- `protected void releaseUserDataForCollection()`

Your code can store and retrieve a user data context object with each row set. This method is called to allow you to release any resources that may be associated with a row set that is being closed.

Since the view object component can be related to several active row sets at runtime, many of the above framework methods receive an `Object` parameter named `qc` in which the framework will pass the collection of rows in question that your code is supposed to be filling, as well as the array of bind variable values that might affect which rows get populated into the specific collection.

You can store a user-data object with each collection of rows so your custom datasource implementation can associate any needed datasource context information. The framework provides the `setUserDataForCollection()` and `getUserDataForCollection()` methods to get and set this per-collection context information. Each time one of the overridden framework methods is called, you can use the `getUserDataForCollection()` method to retrieve the correct `ResultSet` object associated with the collection of rows the framework wants you to populate.

The examples in the following sections each override these methods to implement different kinds of programmatic view objects.

35.9.4 How to Create a View Object on a REF CURSOR

Sometimes your application might need to work with the results of a query that is encapsulated within a stored procedure. PL/SQL allows you to open a cursor to iterate through the results of a query, and then return a reference to this cursor to the client. This so-called `REF CURSOR` is a handle with which the client can then iterate the results of the query. This is possible even though the client never actually issued the original SQL `SELECT` statement.

Declaring a PL/SQL package with a function that returns a `REF CURSOR` is straightforward. For example, your package might look like this:

```
CREATE OR REPLACE PACKAGE RefCursorExample IS
  TYPE ref_cursor IS REF CURSOR;
  FUNCTION get_orders_for_customer(p_email VARCHAR2) RETURN ref_cursor;
  FUNCTION count_orders_for_customer(p_email VARCHAR2) RETURN NUMBER;
END RefCursorExample;
```

After defining an entity-based `OrdersForCustomer` view object with an entity usage for a `Order` entity object, go to its custom Java class

`OrdersForCustomerImpl.java`. At the top of the view object class, define some constant Strings to hold the anonymous blocks of PL/SQL that you'll execute using JDBC `CallableStatement` objects to invoke the stored functions:

```
/*
 * Execute this block to retrieve the REF CURSOR
 */
private static final String SQL =
    "begin ? := RefCursorSample.get_orders_for_customer(?);end;";

/*
 * Execute this block to retrieve the count of orders that
 * would be returned if you executed the statement above.
 */
private static final String COUNTSQL =
    "begin ? := RefCursorSample.count_orders_for_customer(?);end;";
```

Then, override the methods of the view object as described in the following sections.

35.9.4.1 The Overridden `create()` Method

The `create()` method removes all traces of a SQL query for this view object.

```
protected void create() {
    getViewDef().setQuery(null);
    getViewDef().setSelectClause(null);
    setQuery(null);
}
```

35.9.4.2 The Overridden `executeQueryForCollection()` Method

The `executeQueryForCollection()` method calls a helper method `retrieveRefCursor()` to execute the stored function and return the REF CURSOR return value, cast as a JDBC ResultSet. Then, it calls the helper method `storeNewResultSet()` that uses the `setUserDataForCollection()` method to store this ResultSet with the collection of rows for which the framework is asking to execute the query.

```
protected void executeQueryForCollection(Object qc, Object[] params,
                                         int numUserParams) {
    storeNewResultSet(qc, retrieveRefCursor(qc, params));
    super.executeQueryForCollection(qc, params, numUserParams);
}
```

The `retrieveRefCursor()` uses the helper method described in [Section 33.5, "Invoking Stored Procedures and Functions"](#) to invoke the stored function and return the REF CURSOR:

```
private ResultSet retrieveRefCursor(Object qc, Object[] params) {
    ResultSet rs = (ResultSet)callStoredFunction(OracleTypes.CURSOR,
                                                "RefCursorExample.get_requests_for_customer(?)",
                                                new Object[]{getNamedBindParamValue("CustEmail", params)});
    return rs;
}
```

35.9.4.3 The Overridden `createRowFromResultSet()` Method

For each row that the framework needs fetched from the datasource, it will invoke your overridden `createRowFromResultSet()` method. The implementation retrieves the collection-specific ResultSet object from the user-data context, uses the `createNewRowForCollection()` method to create a new blank row in the collection, and then use the `populateAttributeForRow()` method to populate the attribute values for each attribute defined at design time in the View Object Editor.

```
protected ViewRowImpl createRowFromResultSet(Object qc, ResultSet rs) {
    /*
     * We ignore the JDBC ResultSet passed by the framework (null anyway) and
     * use the resultset that we've stored in the query-collection-private
     * user data storage
     */
    rs = getResultSet(qc);

    /*
     * Create a new row to populate
     */
    ViewRowImpl r = createNewRowForCollection(qc);
    try {
        /*
         * Populate new row by attribute slot number for current row in Result Set
         */
        populateAttributeForRow(r, 0, rs.getLong(1));
    }
}
```

```

        populateAttributeForRow(r,1, rs.getString(2));
        populateAttributeForRow(r,2, rs.getString(3));
    }
    catch (SQLException s) {
        throw new JboException(s);
    }
    return r;
}

```

35.9.4.4 The Overridden hasNextForCollectionMethod()

The overridden implementation of the framework method `hasNextForCollection()` has the responsibility to return `true` or `false` based on whether there are more rows to fetch. When you've hit the end, you call the `setFetchCompleteForCollection()` to tell view object that this collection is done being populated.

```

protected boolean hasNextForCollection(Object qc) {
    ResultSet rs = getResultSet(qc);
    boolean nextOne = false;
    try {
        nextOne = rs.next();
        /*
         * When were at the end of the result set, mark the query collection
         * as "FetchComplete".
         */
        if (!nextOne) {
            setFetchCompleteForCollection(qc, true);
            /*
             * Close the result set, we're done with it
             */
            rs.close();
        }
    }
    catch (SQLException s) {
        throw new JboException(s);
    }
    return nextOne;
}

```

35.9.4.5 The Overridden releaseUserDataForCollection() Method

Once the collection is done with its fetch-processing, the overridden `releaseUserDataForCollection()` method gets invoked and closes the `ResultSet` cleanly so no database cursors are left open.

```

protected void releaseUserDataForCollection(Object qc, Object rs) {
    ResultSet userDataRS = getResultSet(qc);
    if (userDataRS != null) {
        try {
            userDataRS.close();
        }
        catch (SQLException s) {
            /* Ignore */
        }
    }
    super.releaseUserDataForCollection(qc, rs);
}

```

35.9.4.6 The Overridden getQueryHitCount() Method

Lastly, in order to properly support the view object's `getEstimatedRowCount()` method, the overridden `getQueryHitCount()` method returns a count of the rows that would be retrieved if all rows were fetched from the row set. Here the code uses a `CallableStatement` to get the job done. Since the query is completely encapsulated behind the stored function API, the code also relies on the PL/SQL package to provide an implementation of the count logic as well to support this functionality.

```
public long getQueryHitCount(ViewRowSetImpl viewRowSet) {
    Object[] params = viewRowSet.getParameters(true);
    BigDecimal id = (BigDecimal)params[0];
    CallableStatement st = null;
    try {
        st = getDBTransaction().createCallableStatement(COUNTSQL,
                                                       DBTransaction.DEFAULT);
        /*
         * Register the first bind parameter as our return value of type CURSOR
         */
        st.registerOutParameter(1, Types.NUMERIC);
        /*
         * Set the value of the 2nd bind variable to pass id as argument
         */
        if (id == null) st.setNull(2,Types.NUMERIC);
        else           st.setBigDecimal(2,id);
        st.execute();
        return st.getLong(1);
    }
    catch (SQLException s) {
        throw new JboException(s);
    }
    finally {try {st.close();} catch (SQLException s) {}}
}
```

35.10 Creating a View Object with Multiple Updatable Entities

Note: To experiment with the example described in this section, use the same `MultipleMasters` project in the `AdvancedViewObjectExamples` workspace.

When you create a view object with multiple entity usages, you can enable a secondary entity usage to be updatable by selecting it in the **Selected** list of the **Entity Objects** page of the view object overview editor and:

- Deselecting the **Reference** checkbox
- Selecting the **Updatable** checkbox

If you only plan to use the view object to update or delete existing data, then this is the only step required. The user can update attributes related to any of the non-reference, updatable entity usages and the view row will delegate the changes to the appropriate underlying entity rows.

However, if you need a view object with multiple updatable entities to support creating new rows and the association between the entity objects is not a composition, then you need to write a bit of code to enable that to work correctly.

Note: You only need to write code to handle creating new rows when the association between the updatable entities is not a composition. If the association is a composition, then ADF Business Components handles this automatically.

When you call `createRow()` on a view object with multiple update entities, it creates new entity row parts for each updatable entity usage. Since the multiple entities in this scenario are related by an association, there are three pieces of code you might need to implement to ensure the new, associated entity rows can be saved without errors:

1. You may need to override the `postChanges()` method on entity objects involved to control the correct posting order.
2. If the primary key of the associated entity is populated by a database sequence using `DBSequence`, and if the multiple entity objects are associated but not composed, then you need to override the `postChanges()` and `refreshFKInNewContainees()` method to handle cascading the refreshed primary key value to the associated rows that were referencing the temporary value.
3. You need to override the `create()` method of the view object's custom view row class to modify the default row creation behavior to pass the context of the parent entity object to the newly-created child entity.

In [Section 34.8, "Controlling Entity Posting Order to Avoid Constraint Violations"](#), you've already seen the code required for 1 and 2 above in an example with associated `Suppliers` and `Products` entity objects. The only thing remaining is the overridden `create()` method on the view row. Consider a `ProductAndSupplier` view object with a primary entity usage of `Product` and secondary entity usages of `Supplier` and `User`. Assume the `Product` entity usage is marked as updatable and non-reference, while the `User` entity usage is a reference entity usage.

[Example 35–30](#) shows the commented code required to correctly sequence the creation of the multiple, updatable entity row parts during a view row create operation.

Example 35–30 Overriding View Row `create()` Method for Multiple Updatable Entities

```
/*
 * By default, the framework will automatically create the new
 * underlying entity object instances that are related to this
 * view object row being created.
 *
 * We override this default view object row creation to explicitly
 * pre-populate the new (detail) ProductsImpl instance using
 * the new (master) SuppliersImpl instance. Since all entity objects
 * implement the AttributeList interface, we can directly pass the
 * new SuppliersImpl instance to the ProductsImpl create()
 * method that accepts an AttributeList.
 */
protected void create(AttributeList attributeList) {
    // The view row will already have created "blank" entity instances
    SuppliersImpl newSupplier = getSupplier();
    ProductsImpl newProduct = getProduct();
    try {
        // Let product "blank" entity instance to do programmatic defaulting
        newSupplier.create(attributeList);
        // Let product "blank" entity instance to do programmatic
        // defaulting passing in new SuppliersImpl instance so its attributes
    }
}
```

```

        // are available to the EmployeeImpl's create method.
        newProduct.create(newSupplier);
    }
    catch (JboException ex) {
        newSupplier.revert();
        newProduct.revert();
        throw ex;
    }
    catch (Exception otherEx) {
        newSupplier.revert();
        newProduct.revert();
        throw new RowCreateException(true      /* EO Row? */,
                                      "Product" /* EO Name */,
                                      otherEx   /* Details */);
    }
}

```

In order for this view row class to be able to invoke the protected `create()` method on the `Suppliers` and `Products` entity objects, they need to override the `create()` method. If the view object and entity objects are in the same package, the overridden `create()` method can have protected access. Otherwise, it requires public access.

```

/**
 * Overriding this method in this class allows friendly access
 * to the create() method by other classes in this same package, like the
 * ProductsAndSuppliers view object implementation class, whose overridden
 * create() method needs to call this.
 * @param nameValuePair
 */
protected void create(Attributelist nameValuePair) {
    super.create(nameValuePair);
}

```

35.11 Declaratively Preventing Insert, Update, and Delete

Some 4GL tools like Oracle Forms provide declarative properties that control whether a given data collection allows inserts, updates, or deletes. While the view object does not yet support this as a built-in feature in the current release, it's easy to add this facility using a framework extension class that exploits custom metadata properties as the developer-supplied flags to control insert, update, or delete on a view object.

To allow developers to have control over individual view object instances, you could adopt the convention of using application module custom properties by the same name as the view object instance. For example, if an application module has view object instances named `ProductsInsertOnly`, `ProductsUpdateOnly`, `ProductsNoDelete`, and `Products`, your generic code might look for application module custom properties by these same names. If the property value contains `Insert`, then insert is enabled for that view object instance. If the property contains `Update`, then update allowed. And, similarly, if the property value contains `Delete`, then delete is allowed. You could use helper methods like this to test for these application module properties and determine whether insert, update, and delete are allowed for a given view object:

```

private boolean isInsertAllowed() {
    return isStringInAppModulePropertyNamedAfterVOInstance("Insert");
}
private boolean isUpdateAllowed() {
    return isStringInAppModulePropertyNamedAfterVOInstance("Update");
}

```

```

private boolean isDeleteAllowed() {
    return isStringInAppModulePropertyNamedAfterVOInstance("Delete");
}
private boolean isStringInAppModulePropertyNamedAfterVOInstance(String s) {
    String voInstName = getViewObject().getName();
    String propVal = (String)getApplicationModule().getProperty(voInstName);
    return propVal != null ? propVal.indexOf(s) >= 0 : true;
}

```

Example 35–31 shows the other code required in a custom framework extension class for view rows to complete the implementation. It overrides the following methods:

- **isAttributeUpdateable()**
To enable the user interface to disable fields in a new row if insert is not allowed or to disable fields in an existing row if update is not allowed.
- **setAttributeInternal()**
To prevent setting attribute values in a new row if insert is not allowed or to prevent setting attributes in an existing row if update is not allowed.
- **remove()**
To prevent remove if delete is not allowed.
- **create()**
To prevent create if insert is not allowed.

Example 35–31 Preventing Insert, Update, or Delete Based on Custom Properties

```

public class CustomViewRowImpl extends ViewRowImpl {
    public boolean isAttributeUpdateable(int index) {
        if (hasEntities() &&
            ((isNewOrInitialized() && !isInsertAllowed()) ||
             (isModifiedOrUnmodified() && !isUpdateAllowed())))
            return false;
        }
        return super.isAttributeUpdateable(index);
    }
    protected void setAttributeInternal(int index, Object val) {
        if (hasEntities()) {
            if (isNewOrInitialized() && !isInsertAllowed())
                throw new JboException("No inserts allowed in this view");
            else if (isModifiedOrUnmodified() && !isUpdateAllowed())
                throw new JboException("No updates allowed in this view");
            }
            super.setAttributeInternal(index, val);
        }
    public void remove() {
        if (!hasEntities() || isDeleteAllowed() || isNewOrInitialized())
            super.remove();
        else
            throw new JboException("Delete not allowed in this view");
    }
    protected void create(AttributeList nvp) {
        if (isInsertAllowed())
            super.create(nvp);
        else {
            throw new JboException("Insert not allowed in this view");
        }
    }
}

```

```
// private helper methods omitted from this example  
}
```


36

Application State Management

This chapter describes the Fusion web application state management facilities and how to use them.

This chapter includes the following sections:

- [Section 36.1, "Understanding Why State Management is Necessary"](#)
- [Section 36.2, "About Fusion Web Application State Management"](#)
- [Section 36.3, "Using Save For Later"](#)
- [Section 36.4, "Setting the Application Module Release Level at Runtime"](#)
- [Section 36.5, "What Model State Is Saved and When It Is Cleaned Up"](#)
- [Section 36.6, "Timing Out the HttpSession"](#)
- [Section 36.7, "Managing Custom User-Specific Information"](#)
- [Section 36.8, "Managing the State of View Objects"](#)
- [Section 36.9, "Using State Management for Middle-Tier Savepoints"](#)
- [Section 36.10, "Testing to Ensure Your Application Module is Activation-Safe"](#)
- [Section 36.11, "Keeping Pending Changes in the Middle Tier"](#)

36.1 Understanding Why State Management is Necessary

Most real-world business applications need to support multi-step user tasks. Modern sites tend to use a step-by-step style user interface to guide the end user through a logical sequence of pages to complete these tasks. When the task is done, the user can save or cancel everything as a unit.

36.1.1 Examples of Multi-Step Tasks

In a typical search-then-edit scenario, the end user searches to find an appropriate row to update, then may open several different pages of related master/detail information to make edits before deciding to save or cancel his work. Consider another scenario where the end user wants to book a vacation online. The process may involve the end user's entering details about:

- One or more flight segments that comprise the journey
- One or more passengers taking the trip
- Seat selections and meal preferences
- One or more hotel rooms in different cities

- Car they will rent

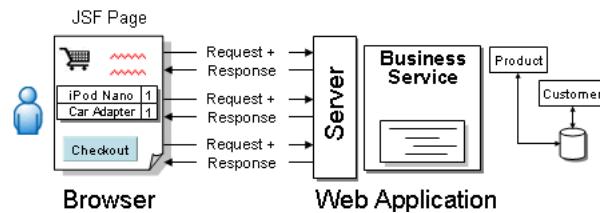
Along the way, the user might decide to complete the transaction, save the reservation for finishing later, or abandoning the whole thing.

It's clear these scenarios involve a logical unit of work that spans multiple web pages. You've seen in previous chapters how to use JDeveloper's JSF page navigation diagram to design the page flow for these use cases, but that is only part of the puzzle. The pending changes the end user makes to business domain objects along the way — Trip, Flight, Passenger, Seat, HotelRoom, Auto, etc. — represent the in-progress state of the application for each end user. Along with this, other types of "bookkeeping" information about selections made in previous steps comprise the complete picture of the application state.

36.1.2 Stateless HTTP Protocol Complicates Stateful Applications

While it may be easy to imagine these multi-step scenarios, *implementing* them in web applications is complicated by the stateless nature of HTTP, the hypertext transfer protocol. [Figure 36–1](#) illustrates how an end user's visit to a site comprises a series of HTTP request/response pairs. However, HTTP affords a web server no way to distinguish one user's request from another user's, or to differentiate between a single user's first request and any subsequent requests he makes while interacting with the site. The server gets each request from any user always as if it were the first (and only) one they make.

Figure 36–1 Web Applications Use the Stateless HTTP Protocol



But even if you've never *implemented* your own web applications before, since you've undoubtedly *used* a web application to buy a book, plan a holiday, or even just read your email, it's clear that a solution must exist to distinguish one user from another.

36.1.3 How Cookies Are Used to Track a User Session

As shown in [Figure 36–2](#), the technique used to recognize an ongoing sequence of requests from the same end user over the stateless HTTP protocol involves a unique identifier called a *cookie*. A cookie is a name/value pair that is sent in the header information of each HTTP request the user makes to a site. On the initial request made by a user, the cookie is not part of the request. The server uses the *absence* of the cookie to detect the start of a user's session of interactions with the site, and it returns a unique identifier to the browser that represents this session for this user. In practice, the cookie value is a long string of letters and numbers, but for the simplicity of the illustration, assume that the unique identifier is a letter like "A" or "Z" that corresponds to different users using the site.

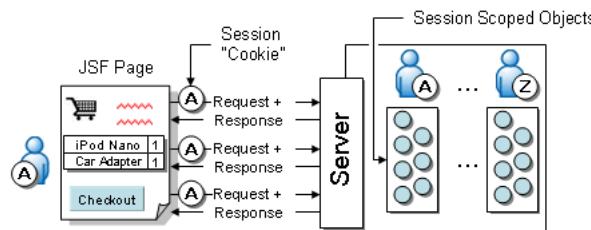
Web browsers support a standard way of recognizing the cookie returned by the server that allows the browser to identify the following:

- the site that sent the cookie
- how long it should remember the cookie value

On each subsequent request made by that user, until the cookie expires, the browser sends the cookie along in the header of the request. The server uses the value of the cookie to distinguish between requests made by different users.

A cookie that expires when you close your browser is known as a session cookie, while other cookies that are set to live beyond a single browser session might expire in a week, a month, or a year from when they were first created.

Figure 36–2 Tracking State Using a Session Cookies and Server-Side Session



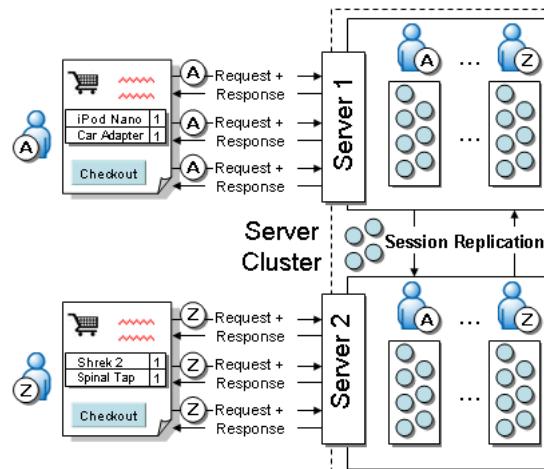
Java EE-compliant web servers provide a standard server-side facility called the `HttpSession` that allows a web application to store Java objects related to a particular user's session as named attribute/value pairs. An object placed in this session Map on one request can be retrieved by the application while handling a subsequent request during the same session.

The session remains active while the user continues to send new requests within the timeframe configured by the `<session-timeout>` element in the `web.xml` file. The default session length is 35 minutes.

36.1.4 Performance and Reliability Impact of Using HttpSession

The `HttpSession` facility is an ingredient in most application state management strategies, but it can present performance and reliability problems if not used judiciously. First, because the session-scope Java objects you create are held in the memory of the Java EE web server, the objects in the HTTP session are lost if the server should fail.

As shown in [Figure 36–3](#), one way to improve the reliability is to configure multiple Java EE servers in a cluster. By doing this, the Java EE application server replicates the objects in the HTTP session for each user across multiple servers in the cluster so that if one server goes down, the objects exist in the memory of the other servers in the cluster that can continue to handle the user's requests. Since the cluster comprises separate servers, replicating the HTTP session contents among them involves broadcasting the changes made to HTTP session objects over the network.

Figure 36–3 Session Replication in a Server Cluster

You can begin to see some of the performance implications of overusing the HTTP session:

- The more active users, the more HTTP sessions will be created on the server.
- The more objects stored in each HTTP session, the more memory you will need. Note that the memory is not reclaimed when the user becomes inactive; this only happens with a session timeout or an explicit session invalidation. Session invalidations don't always happen because users don't always logout.
- In a cluster, the more objects in each HTTP session that *change*, the more network traffic will be generated to replicate the changed objects to other servers in the cluster.

At the outset, it would seem that keeping the number of objects stored in the session to a minimum addresses the problem. However, this implies leveraging an alternative mechanism for temporary storage for each user's pending application state. The most popular alternatives involve saving the application state to the database between requests or to a file of some kind on a shared file system.

Of course, this is easier said than done. A possible approach involves eagerly saving the pending changes to your underlying database tables and committing the transaction at the end of each HTTP request. But this idea has two key drawbacks:

- *Your database constraints might fail.*
At any given step of the multi-step process, the information may only be partially complete, and this could cause errors at the database level when trying to save the changes.
- *You complicate rolling back the changes.*
Cancelling the logical unit of work would involve carefully deleting all of the eagerly-committed rows in possible multiple tables.

These limitations have lead developers in the past to invent solutions involving a "shadow" set of database tables with no constraints and with all of the column types defined as character-based. Using such a solution becomes very complex very quickly. Ultimately, you will conclude that you need some kind of generic application state management facility to address these issues in a more generic and workable way. The solution comes in the form of ADF Business Components, which implements this for you out of the box.

36.2 About Fusion Web Application State Management

State management enables you to easily create web applications that support multi-step use cases without falling prey to the memory, reliability, or implementation complexity problems described in [Section 36.1, "Understanding Why State Management is Necessary"](#).

Application state management is provided at two levels, by the Save For Later feature in a task flow, and application module state management in the model layer.

Save For Later is activated at the controller layer and automatically saves a "snapshot" of the current UI and controller states, and delegates to the model layer to *passivate* (save) its state as well.

If you are not using ADF data controls, you can still use application module state management alone, but since this will save only the model state, this is an outside case for most applications.

36.2.1 Basic Architecture of the Save for Later Facility

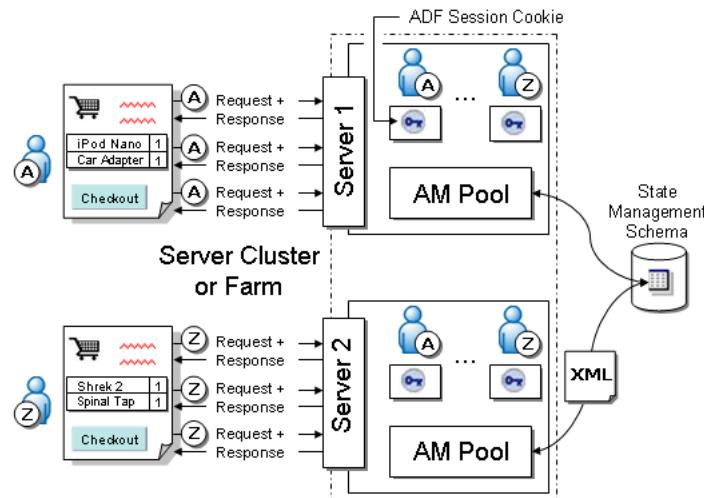
Save for Later saves an incomplete transaction without enforcing validation rules or submitting the data. The end user can resume working on the same transaction later with the same data that was originally saved when the application was exited.

36.2.2 Basic Architecture of the Application Module State Management Facility

Your ADF Business Components-based application automatically manages the application state of each user session. This provides the simplicity of a stateful programming model that you are accustomed to in previous 4GL tools, yet implemented in a way that delivers scalability nearing that of a purely stateless application. Understanding what happens behind the scenes is essential to make the most efficient use of this important feature.

You can use application module components to implement completely stateless applications or to support a unit of work that spans multiple browser pages.

[Figure 36-4](#) illustrates the basic architecture of the state management facility to support these multi-step scenarios. An application module supports *passivating* (storing) its pending transaction state to an XML document, which is stored in the database in a single, generic table, keyed by a unique passivation snapshot ID. It also supports the reverse operation of *activating* pending transaction state from one of these saved XML snapshots. This passivation and activation is performed automatically by the application module pool when needed.

Figure 36–4 ADF Provides Generic, Database-Backed State Management

The ADF binding context is the one object that lives in the `HttpSession` for each end user. It holds references to lightweight application module data control objects that manage acquiring an application module instance from the pool during the request (when the data control is accessed) and releasing it to the pool at the end of each request. The data control holds a reference to the ADF session cookie that identifies the user session. In particular, business domain objects created or modified in the pending transaction are *not* saved in the `HttpSession` using this approach. This minimizes both the session memory required per user and eliminates the network traffic related to session replication if the servers are configured in a cluster.

For improved reliability, if you have multiple application servers and you enable the optional ADF Business Components failover support (explained in [Section 36.2.2.2, "How Passivation Changes When Optional Failover Mode is Enabled"](#)), then subsequent end-user requests can be handled by any server in your server farm or cluster. The session cookie can reactivate the pending application state from the database-backed XML snapshot if required, regardless of which server handles the request.

36.2.2.1 Understanding When Passivation and Activation Occurs

To better understand when the automatic passivation and activation of application module state occurs, consider the following simple case:

- At the beginning of an HTTP request, the application module data control handles the `beginRequest` event by checking out an application module instance from the pool.

The application module pool returns an *unreferenced* instance. An unreferenced application module is one that is not currently managing the pending state for any other user session.

- At the end of the request, the application module data control handles the `endRequest` event by checking the application module instance back into the pool in "managed state" mode.

That application module instance is now *referenced* by the data control that just used it. And the application module instance is an object that still contains pending transaction state made by the data control (that is, entity object and view object caches; updates made but not committed; and cursor states), stored in

memory. As you'll see below, it's not *dedicated* to this data control, just referenced by it.

3. On a subsequent request, the same data control — identified by its `SessionCookie` — checks out an application module instance again.

Due to the "stateless with user affinity" algorithm the pool uses, you might assume that the pool returns the exact same application module instance, with the state still there in memory. (To understand this algorithm, read [Section 37.1, "Overview of Application Module Pooling"](#) and the discussion of Referenced Pool Size in [Section 37.5.2, "Pool Sizing Parameters"](#).)

Sometimes due to a high number of users simultaneously accessing the site, application module instances must be sequentially reused by different user sessions. In this case, the application pool must *recycle* a currently referenced application module instance for use by another session, as follows:

1. The application module data control for User A's session checks an application module instance into the application pool at the end of a request. Assume this instance is named AM1.
2. The application module data control for User Z's new session requests an application module instance from the pool for the first time, but there are no unreferenced instances available. The application module pool then:
 - Passivates the state of instance AM1 to the database.
 - Resets the state of AM1 in preparation to be used by another session.
 - Returns the AM1 instance to User Z's data control.
3. On a subsequent request, the application module data control for User A's session requests an application module instance from the pool. The application module pool then:
 - Obtains an unreference instance.

This could be instance AM1, obtained by following the same steps as in (2) above, or another AM2 instance if it had become unreferenced in the meantime.

 - Activates the appropriate pending state for User A from the database.
 - Returns the application module instance to User A's data control.

The process of passivation, activation, and recycling allows the state referenced by the data control to be preserved across requests without requiring a dedicated application module instance for each data control. Both browser users in the above scenario are carrying on an application transaction that spans multiple HTTP requests, but the end users are unaware whether the passivation and activation is occurring in the background. They just continue to see the pending changes. In the process, the pending changes never need to be saved into the underlying application database tables until the end user is ready to commit the logical unit of work.

Note that this keeps the session memory footprint low because the only business component objects that are directly referenced by the session (and are replicable) are the data control and the session cookie.

The application module pool makes a best effort to keep an application module instance "sticky" to the current data control whose pending state it is managing. This is known as maintaining user session affinity. The best performance is achieved if a data control continues to use exactly the same application module instance on each request, since this avoids any overhead involved in reactivating the pending state from a persisted snapshot.

36.2.2.2 How Passivation Changes When Optional Failover Mode is Enabled

The `jbo.dofailover` parameter controls when and how often passivation occurs. You can set this parameter in your application module configuration on the **Pooling and Scalability** tab of the Configuration Editor. When the failover feature is disabled, which it is by default, then application module pending state will only be passivated on demand when it must be. This occurs just before the pool determines it must hand out a currently-referenced application module instance to a *different* data control.

Note: Passivation can also occur when an application module is timed out. For more information about application pool removal algorithms (such as `jbo.ampool.timetolive`), see [Section 37.5.3, "Pool Cleanup Parameters"](#).

In contrast, with the failover feature turned on, the application module's pending state is passivated every time it is checked back into application module pool. This provides the most pessimistic protection against application server failure. The application module instances' state is always saved and may be activated by any application module instance at any time. Of course, this capability comes at expense of the additional overhead of eager passivation on each request.

Note: When running or debugging an application that uses failover support within the JDeveloper environment, you are frequently starting and stopping the application server. The ADF failover mechanism has no way of knowing whether you stopped the server to simulate an application server failure, or whether you stopped it because you want to retest something from scratch in a fresh server instance. If you intend to do the latter, exit out of your browser before restarting the application on the server. This eliminates the chance that you will be confused by the correct functioning of the failover mechanism when you didn't intend to be testing that aspect of your application.

36.2.2.3 About State Management Release Levels

When a data control handles the `endRequest` notification indicating the processing for the current HTTP request has completed, it releases the application module instance by checking it back into the application module pool. The application module pool manages instances and performs state management tasks (or not) based on the *release level* you use when returning the instance to the pool.

There are three release levels used for returning an instance of an application module to a pool:

- Managed - This is the default level, where the application module pool prefers to keep the same application module instance for the same data control, but may release an instance if necessary.
- Unmanaged - No state needs to be preserved beyond the current request.
- Reserved - A one-to-one relationship is preserved between an application module instance and a data control.

Caution: In general, it is strongly recommended never to use Reserved release level. You would normally avoid using this mode because the data control to application module correlation becomes one to one, the scalability of the application reduces very sharply, and so does reliability of the application.

36.2.2.3.1 About Managed Release Level This is the default release level and implies that application module's state is relevant and has to be preserved for this data control to span over several HTTP requests. Managed level does not guarantee that for the next request this data control will receive the same physical application module instance, but it does guarantees that an application module with identical state will be provided so it is logically the same application module instance each time. It is important to note that the framework makes the best effort it can to provide the same instance of application module for the same data control if it is available at the moment. This is done for better performance since the same application module does not need to activate the previous state which it still has intact after servicing the same data control during previous request. However, the data control is not guaranteed to receive the same instance for all its requests and if the application module that serviced that data control during previous is busy or unavailable, then a different application module will activate this data control's state. For this reason, it is not valid to cache references to application module objects, view objects, or view rows across HTTP requests in controller-layer code.

This mode was called the "Stateful Release Mode" in previous releases of JDeveloper.

Note: If the `jbo.ampool.doampooling` configuration property is `false` — corresponding to your unchecking the **Enable Application Module Pooling** option in the **Pooling and Scalability** tab of the Configuration Editor — then there is effectively no pool. In this case, when the application module instance is released at the end of a request it is immediately removed. On subsequent requests made by the same user session, a new application module instance must be created to handle each user request, and pending state must be reactivated from the passivation store. Setting this property to `false` is useful to discover problems in your application logic that might occur when reactivation does occur due to unpredictable load on your system. However, the property `jbo.ampool.doampooling` set to `false` is not a supported configuration for production applications and must be set to `true` before you deploy your application. For further details, see [Section 36.10, "Testing to Ensure Your Application Module is Activation-Safe"](#).

36.2.2.3.2 About Unmanaged Release Level This mode implies that no state associated with this data control has to be preserved to survive beyond the current HTTP request. This level is the most efficient in performance because there is no overhead related to state management. However, you should limit its use to applications that require no state management, or to cases when state no longer needs to be preserved at this point. Usually, you can programmatically release the application module with the unmanaged level when you want to signal that the user has ended a logical unit of work.

Performance Tip: .The default release level is Managed, which implies that the application module's state is relevant and has to be preserved to allow the data control to span over several HTTP requests. Set release level to Unmanaged programmatically at runtime for particular pages to eliminate passivation and achieve better performance. A typical example is releasing the application module after servicing the HTTP request from a logout page.

This mode was called the "Stateless Release Mode" in previous releases of JDeveloper.

36.2.2.3.3 About Reserved Release Level This level guarantees that each data control will be assigned its own application module during its first request and for all subsequent requests coming from the HttpSession associated with this data control. This data control will always receive the same physical instance of application module. This mode exists for legacy compatibility reasons and for very rare special use cases.

An example of using Reserved level occurs when there is a pending database state across a request resulting from the `postChanges()` method or a PL/SQL stored procedure but not issuing a `commit()` or `rollback()` at the end of the request. In this case, if any other release level is used instead of Reserved, when the application module instance is recycled, a rollback is issued on the database connection associated with this application module instance and all uncommitted changes would be lost.

Performance Tip: If you must use Reserved level, call `setReleaseLevel()` on the data control to keep its period as short as possible. For details about changing the release level programmatically, see [Section 36.4, "Setting the Application Module Release Level at Runtime"](#).

Consequences of Reserved mode can be adverse. Reliability suffers because if for whatever reason the application module is lost, the data control will not be able to receive any other application module in its place from the pool, and so HttpSession gets lost as well, which is not the case for managed level.

The failover option is ignored for an application module released with Reserved release level since its use implies your application absolutely requires working with the same application module instance on each request.

36.3 Using Save For Later

To enable Save For Later, you must first add Save Points to the application at points where you would like application state and data to be preserved if the end user leaves the application. You can use it to save data and state information about a region, view port, or portlet. Later, you use the Save Point Restore activity to restore application state and data associated with a Save Point.

For more information on how create and restore Save Points, see [Section 17.5, "Saving for Later"](#).

Save For Later can also perform implicit saves. These occur when data is saved automatically without the end user performing an explicit Save action when the user session times out or closes the browser window, for example.

For more information on how to perform an implicit save, see [Section 17.5, "Saving for Later"](#).

36.4 Setting the Application Module Release Level at Runtime

If you do not want to use the default "Managed State" release level for application modules, you can set your desired level programmatically.

36.4.1 How to Set Unmanaged Level

To set a data control to release its application module using the unmanaged level, call the `resetState()` method on the `DCDataControl` class (in the `oracle.adf.model.binding` package).

You can call this method any time during the request. This will cause application module not to passivate its state at all when it is released to the pool at the end of the request. Note that this method only affects the current application module instance in the current request. After this, the application module is released in unmanaged level to the pool, it becomes unreferenced and gets reset. The next time the application module is used by a client, it will be used in the managed level again by default.

Note: You can programmatically release the application module with the unmanaged level when you want to signal that the user has ended a logical unit of work. This will happen automatically when the `HTTPSession` times out, as described below.

36.4.2 How to Set Reserved Level

To set a data control to release its application module using the reserved level, call the `setReleaseLevel()` method of the `DCJboDataControl` class (in the `oracle.adf.model.bc4j` package), and pass the integer constant `ApplicationModule.RELEASE_LEVEL_RESERVED`.

When the release level for an application module has been changed to "Reserved" it will stay so for all subsequent requests until explicitly changed.

36.4.3 How to Set Managed Level

If you have set an application module to use reserved level, you can later set it back to use managed level by calling the `setReleaseLevel()` method of the `DCJboDataControl` class, and passing the integer constant `ApplicationModule.RELEASE_LEVEL_MANAGED`.

36.4.4 How to Set Release Level in a JSF Backing Bean

[Example 36–1](#) shows calling the `resetState()` method on a data control named `UserModuleDataControl` from the action method of a JSF backing bean.

Example 36–1 Calling `resetState()` on Data Control in a JSF Backing Bean Action Method

```
package devguide.advanced.releasestateless.controller.backing;
import devguide.advanced.releasestateless.controller.JSFUtils;
import oracle.adf.model.BindingContext;
import oracle.adf.model.binding.DCDataControl;
/**
 * JSF Backing bean for the "Example.jspx" page
 */
public class Example {
    /**
     * In an action method, call resetState() on the data control to cause
```

```
* it to release to the pool with the "unmanaged" release level.  
* In other words, as a stateless application module.  
*/  
public String commandButton_action() {  
    // Add event code here...  
    getDataControl("UserModuleDataControl").resetState();  
    return null;  
}  
private DCDataControl getDataControl(String name) {  
    BindingContext bc =  
        (BindingContext)JSFUtils.resolveExpression("#{data}");  
    return bc.findDataControl(name);  
}  
}
```

36.4.5 How to Set Release Level in an ADF PagePhaseListener

[Example 36–2](#) shows calling the `resetState()` method on a data control named `UserModuleDataControl` from the after-prepareRender phase of the ADF lifecycle using a custom ADF page phase-listener class. You would associate this custom class to a particular page by setting the `ControllerClass` attribute on the page's page definition to the fully-qualified name of this class.

Example 36–2 Calling `resetState()` on Data Control in a Custom PagePhaseListener

```
package devguide.advanced.releasestateless.controller;  
import oracle.adf.controller.v2.lifecycle.Lifecycle;  
import oracle.adf.controller.v2.lifecycle.PagePhaseEvent;  
import oracle.adf.controller.v2.lifecycle.PagePhaseListener;  
import oracle.adf.model.binding.DCDataControl;  
public class ReleaseStatelessPagePhaseListener  
    implements PagePhaseListener {  
    /**  
     * In the "after" phase of the final "prepareRender" ADF Lifecycle  
     * phase, call resetState() on the data control to cause it to release  
     * to the pool with the "unmanaged" release level. In other words,  
     * as a stateless application module.  
     *  
     * @param event ADF page phase event  
     */  
    public void afterPhase(PagePhaseEvent event) {  
        if (event.getPhaseId() == Lifecycle.PREPARE_RENDER_ID) {  
            getDataControl("UserModuleDataControl", event).resetState();  
        }  
    }  
    // Required to implement the PagePhaseListener interface  
    public void beforePhase(PagePhaseEvent event) {}  
    private DCDataControl getDataControl(String name,  
                                         PagePhaseEvent event) {  
        return event.getLifecycleContext()  
            .getBindingContext()  
            .findDataControl(name);  
    }  
}
```

36.4.6 How to Set Release Level in an ADF PageController

[Example 36–3](#) shows calling the `resetState()` method on a data control named `UserModuleDataControl` from an overridden `prepareRender()` method of a custom

ADF page controller class. You would associate this custom class to a particular page by setting the `ControllerClass` attribute on the page's page definition to the fully-qualified name of this class.

Note: You can accomplish basically the same kinds of page-specific lifecycle customization tasks using a custom `PagePhaseListener` or a custom `PageController` class. The key difference is that the `PagePhaseListener` interface can be implemented on any class, while a custom `PageController` must extend the `PageController` class in the `oracle.adf.controller.v2.lifecycle` package.

Example 36–3 Calling `resetState()` on Data Control in a Custom ADF PageController

```
package devguide.advanced.releasestateless.controller;
import oracle.adf.controller.v2.context.LifecycleContext;
import oracle.adf.controller.v2.lifecycle.PageController;
import oracle.adf.controller.v2.lifecycle.PagePhaseEvent;
import oracle.adf.model.binding.DCDataControl;
public class ReleaseStatelessPageController extends PageController {
    /**
     * After calling the super in the final prepareRender() phase
     * of the ADF Lifecycle, call resetState() on the data control
     * to cause it to release to the pool with the "unmanaged"
     * release level. In other words, as a stateless application module.
     *
     * @param lcCtx ADF lifecycle context
     */
    public void prepareRender(LifecycleContext lcCtx) {
        super.prepareRender(lcCtx);
        getDataControl("UserModuleDataControl", lcCtx).resetState();
    }
    private DCDataControl getDataControl(String name,
                                         LifecycleContext lcCtx) {
        return lcCtx.getBindingContext().findDataControl(name);
    }
}
```

36.4.7 How to Set Release Level in an Custom ADF PageLifecycle

If you wanted to build a Fusion web application where every request was handled in a completely stateless way, use a global custom `PageLifecycle` class as shown in [Example 36–4](#). For details on how to configure your application to use your custom lifecycle see [Section 19.2, "The JSF and ADF Page Lifecycles"](#).

Example 36–4 Calling `resetState()` on Data Control in a Custom ADF PageLifecycle

```
package devguide.advanced.releasestateless.controller;
import oracle.adf.controller.faces.lifecycle.FacesPageLifecycle;
import oracle.adf.controller.v2.context.LifecycleContext;
import oracle.adf.model.binding.DCDataControl;
public class ReleaseStatelessPageLifecycle extends FacesPageLifecycle {
    /**
     * After calling the super in the final prepareRender() phase
     * of the ADF Lifecycle, call resetState() on the data control
     * to cause it to release to the pool with the "unmanaged"
     * release level. In other words, as a stateless application module.
     *
     * @param lcCtx ADF lifecycle context
     */
}
```

```
/*
public void prepareRender(LifecycleContext lcCtx) {
    super.prepareRender(lcCtx);
    getDataControl("UserModuleDataControl", lcCtx).resetState();
}
private DCDataControl getDataControl(String name,
                                     LifecycleContext lcCtx) {
    return lcCtx.getBindingContext().findDataControl(name);
}
}
```

36.5 What Model State Is Saved and When It Is Cleaned Up

The information saved by application model passivation is divided in two parts: transactional and non-transactional state. Transactional state is the set of updates made to entity object data – performed either directly on entity objects or on entities through view object rows – that are intended to be saved into the database. Non-transactional state comprises view object runtime settings, such as the current row index, WHERE clause, and ORDER BY clause.

36.5.1 What State is Saved?

The information saved as part of the application module passivation "snapshot" includes the following.

Transactional State

- New, modified, and deleted entities in the entity caches of the root application module for this user session's (including old/new values for modified ones).

Non-Transactional State

- For each active view object (both statically and dynamically created):
 - Current row indicator for each row set (typically one)
 - New rows and their positions (New rows are treated differently than updated ones. Their index in the VO is traced as well)
 - ViewCriteria and all its related parameters such as view criteria row etc.
 - Flag indicating whether or not a row set has been executed
 - Range start and Range size
 - Access mode
 - Fetch mode and fetch size
 - Any view object-level custom data
 - SELECT, FROM, WHERE, and ORDER BY clause if created dynamically or changed from the View definition

Note: If you enable ADF Business Components runtime diagnostics, the contents of each XML state snapshot. See [Section 6.3.7, "How to Enable ADF Business Components Debug Diagnostics"](#) for more information on how to enable diagnostics.

36.5.2 Where is the Model State Saved?

By default, passivation snapshots are saved in the database, but you can configure it to use the file system as an alternative.

36.5.2.1 How Database-Backed Passivation Works

The passivated XML snapshot is written to a BLOB column in a table named PS_TXN, using a connection specified by the jbo.server.internal_connection property. Each time a passivation record is saved, it is assigned a unique passivation snapshot ID based on the sequence number taken from the PS_TXN_SEQ sequence. The ADF session cookie held by the application module data control in the ADF binding context remembers the latest passivation snapshot ID that was created on its behalf and remembers the previous ID that was used.

36.5.2.2 Controlling the Schema Where the State Management Table Resides

The ADF runtime recognizes a configuration property named jbo.server.internal_connection that controls which database connection/schema should be used for the creation of the PS_TXN table and the PS_TXN_SEQ sequence. If you don't set the value of this configuration parameter explicitly, then the state management facility creates the temporary tables using the credentials of the current application database connection.

To keep the temporary information separate, the state management facility will use a different connection *instance* from the database connection pool, but the database credentials will be the same as the current user. Since the framework creates temporary tables, and possibly a sequence if they don't already exists, the implication of not setting a value for the jbo.server.internal_connection is that the current database user must have CREATE TABLE, CREATE INDEX and CREATE SEQUENCE privileges. Since this is often not desirable, Oracle recommends always supplying an appropriate value for the jbo.server.internal_connection property, providing the credentials for a "state management" schema where table and schema be created. Valid values for the jbo.server.internal_connection property in your configuration are:

- A fully-qualified JDBC connection URL like:
`jdbc:oracle:thin:username/password@host:port:SID`
- A JDBC datasource name like:
`java:/comp/env/jdbc/YourJavaEEDataSourceName`

36.5.2.3 Configuring the Type of Passivation Store

Passivated information can be stored in several places. You can control it programmatically or by configuring an option in the application module configuration. The choices are database or a file stored on local file system:

- **File**

This choice may be the fastest available as access to the file is faster then access to the database. This choice is good if the entire middle tier (one or multiple Oracle Application Server installation(s) and all their server instances) is either installed on the same machine or has access to a commonly shared file system, so passivated information is accessible to all. Usually, this choice may be good for a small middle tier where one Oracle Application Server is used. In other words this is very suitable choice for small middle tier such as one Oracle Application Server with all its components installed on one physical machine. The location and name of the persistent snapshot files are determined by jbo.tmpdir property if

specified. It follows usual rules of ADF property precedence for a configuration property. If nothing else is specified, then the location is determined by `user.dir` if specified. This is a default property and the property is OS specific.

- **Database**

This is the *default* choice. While it may be a little slower than passivating to file, it is by far the most reliable choice. With passivation to file, the common problem might be that it is not accessible to Oracle Application Server instances that are remotely installed. In this case, in a cluster environment, if one node goes down the other may not be able to access passivated information and then failover will not work. Another possible problem is that even if file is accessible to the remote node, the access time for the local and remote node may be very different and performance will be inconsistent. With database access, time should be about the same for all nodes.

To set the value of your choice in design time, set the property `jbo.passivationstore` to `database` or `file`. The value `null` will indicate that a connection-type-specific default should be used. This will use database passivation for Oracle or DB2, and file serialization for any others.

To set the storage programmatically use the method `setStoreForPassiveState()` of interface `oracle.jbo.ApplicationModule`. The parameter values that you can pass are:

- `PASSIVATE_TO_DATABASE`
- `PASSIVATE_TO_FILE`

36.5.3 When is the Model State Cleaned Up?

Under normal circumstances, the ADF state management facility provides automatic cleanup of the passivation snapshot records.

36.5.3.1 Previous Snapshot Removed When Next One Taken

When a passivation record is saved to the database on behalf of a session cookie, as described above, this passivation record gets a new, unique snapshot ID. The passivation record with the previous snapshot ID used by that same session cookie is deleted as part of the same transaction. In this way, assuming no server failures, there will only ever be a single passivation snapshot record per active end-user session.

36.5.3.2 Passivation Snapshot Removed on Unmanaged Release

The passivation snapshot record related to a session cookie is removed when the application module is checked into the pool with the unmanaged state level. This can occur when:

- Your code specifically calls `resetState()` on the application module data control.
- Your code explicitly invalidates the `HttpSession`, for example, as part of implementing an explicit "Logout" functionality.
- The `HttpSession` times out due to exceeding the session timeout threshold for idle time and failover mode is disabled (which is the default).

In each of these cases, the application module pool also resets the application module referenced by the session cookie to be "unreferenced" again. Since no changes were ever saved into the underlying database tables, once the pending session state

snapshots are removed, there remains no trace of the unfinished work the user session had completed up to that point.

36.5.3.3 Passivation Snapshot Retained in Failover Mode

When the failover mode is enabled, if the HttpSession times out due to session inactivity, then the passivation snapshot is retained so that the end user can resume work upon returning to the browser.

After a break in the action, when the end user returns to his browser and continues to use the application, it continues working as if nothing had changed. The session cookie is used to reactivate any available application module instance with the user's last pending state snapshot before handling the request. So, even though the users next request will be processed in the context of a new HttpSession (perhaps even in a different application server instance), the user is unaware that this has occurred.

Note: If an application module was released with reserved level then the HttpSession times out, the user will have to go through authentication process, and all unsaved changes are lost.

36.5.4 Cleaning Up Temporary Storage Tables

JDeveloper supplies the `bc4jcleanup.sql` script in the `./BC4J/bin` directory to help with periodically cleaning up the application module state management table. Persistent snapshot records can accumulate over time if the server has been shutdown in an abnormal way, such as might occur during development or due to a server failure. Running the script in SQL*Plus will create the `BC4J_CLEANUP` PL/SQL package. The two relevant procedures in this package are:

- PROCEDURE `Session_State(olderThan DATE)`
This procedure cleans-up application module session state storage for sessions older than a given date.
- PROCEDURE `Session_State(olderThan_minutes INTEGER)`
This procedures cleans-up application module session state storage for sessions older than a given number of minutes.

You can schedule periodic cleanup of your ADF temporary persistence storage by submitting an invocation of the appropriate procedure in this package as a database job.

You can use an anonymous PL/SQL block like the one shown in [Example 36–5](#) to schedule the execution of `bc4j_cleanup.session_state()` to run starting tomorrow at 2:00am and each day thereafter to cleanup sessions whose state is over 1 day (1440 minutes) old.

Example 36–5 Scheduling Periodic Cleanup of the State Management Table

```
SET SERVEROUTPUT ON
DECLARE
    jobId      BINARY_INTEGER;
    firstRun DATE;
BEGIN
    -- Start the job tomorrow at 2am
    firstRun := TO_DATE(TO_CHAR(SYSDATE+1, 'DD-MON-YYYY') || ' 02:00',
                        'DD-MON-YYYY HH24:MI');
    -- Submit the job, indicating it should repeat once a day
    dbms_job.submit(job        => jobId,
```

```

-- Run the BC4J Cleanup for Session State
-- to cleanup sessions older than 1 day (1440 minutes)
what      => 'bc4j_cleanup.session_state(1440);',
next_date => firstRun,
-- When completed, automatically reschedule
-- for 1 day later
interval   => 'SYSDATE + 1'
);
dbms_output.put_line('Successfully submitted job. Job Id is'||jobId);
END;
.
/

```

36.6 Timing Out the HttpSession

Since HTTP is a stateless protocol, the server receives no implicit notice that a client has closed his browser or gone away for the weekend. Therefore any Java EE-compliant server provides a standard, configurable session timeout mechanism to allow resources tied to the HTTP session to be freed when the user has stopped performing requests. You can also programmatically force a timeout.

36.6.1 How to Configure the Implicit Timeout Due to User Inactivity

You configure the session timeout threshold using the `<session-timeout>` tag in the `web.xml` file. The default value is 35 minutes. When the `HttpSession` times out the `BindingContext` goes out of scope, and along with it, any data controls that might have referenced application modules released to the pool in the managed state level. The application module pool resets any of these referenced application modules and marks the instances unreferenced again.

36.6.2 How to Code an Explicit HttpSession Timeout

To end a user's session before the session timeout expires, you can call the `invalidate()` method on the `HttpSession` object from a backing bean in response to the user's click on a Logout button or link. This cleans up the `HttpSession` in the same way as if the session time had expired. Using JSF and ADF, after invalidating the session, you *must* perform a redirect to the next page you want to display, rather than just doing a forward. [Example 36–6](#) shows sample code to perform this task from a Logout button.

Example 36–6 Programmatically terminating a session

```

public String logoutButton_action() throws IOException{
    ExternalContext ectx = FacesContext.getCurrentInstance().getExternalContext();
    HttpServletResponse response = (HttpServletResponse)ectx.getResponse();
    HttpSession session = (HttpSession)ectx.getSession(false);
    session.invalidate();
    response.sendRedirect("Welcome.jsp");
    return null;
}

```

As with the implicit timeouts, when the HTTP session is cleaned up this way, it ends up causing any referenced application modules to be marked unreferenced.

36.7 Managing Custom User-Specific Information

It is fairly common practice to add custom user-defined information in the application module in the form of member variables or some custom information stored in `oracle.jbo.Session` user data hashtable. The ADF state management facility provides a mechanism to save this custom information to the passivation snapshot as well, by overriding the `passivateState()` method and `activateState()` method in the `ApplicationModuleImpl` class.

Note: Similar methods are available on the `ViewObjectImpl` class and the `EntityObjectImpl` class to save custom state for those objects to the passivation snapshot as well.

36.7.1 How to Passivate Custom User-Specific Information

[Example 36–7](#) shows how to override `passivateState()` and `activateState()` to ensure custom application module state information is included in the passivation/activation cycle.

In the example, `jbo.counter` contains custom values you want to preserve across passivation and activation of the application module state. Each application module has an `oracle.jbo.Session` object associated with it that stores application module-specific session-level state. The session contains a user data hashtable where you can store transient information. For the user-specific data to "survive" across application module passivation and reactivation, you need to write code to save and restore this custom value into the application module state passivation snapshot.

Example 36–7 Passivating and Activating Custom Information in the State Snapshot XML Document

```
/**
 * Overridden framework method to passivate custom XML elements
 * into the pending state snapshot document
 */
public void passivateState(Document doc, Element parent) {
    // 1. Retrieve the value of the value to save
    int counterValue = getCounterValue();
    // 2. Create an XML element to contain the value
    Node node = doc.createElement(COUNTER);
    // 3. Create an XML text node to represent the value
    Node cNode = doc.createTextNode(Integer.toString(counterValue));
    // 4. Append the text node as a child of the element
    node.appendChild(cNode);
    // 5. Append the element to the parent element passed in
    parent.appendChild(node);
}
/**
 * Overridden framework method to activate custom XML elements
 * into the pending state snapshot document
 */
public void activateState(Element elem) {
    super.activateState(elem);
    if (elem != null) {
        // 1. Search the element for any <jbo.counter> elements
        NodeList nl = elem.getElementsByTagName(COUNTER);
        if (nl != null) {
            // 2. If any found, loop over the nodes found
            for (int i=0, length = nl.getLength(); i < length; i++) {
```

```

        // 3. Get first child node of the <jbo.counter> element
        Node child = nl.item(i).getFirstChild();
        if (child != null) {
            // 4. Set the counter value to the activated value
            setCounterValue(new Integer(child.getNodeValue()).intValue() + 1);
            break;
        }
    }
}
/*
 * Helper Methods
 */
private int getCounterValue() {
    String counterValue = (String)getSession().getUserData().get(COUNTER);
    return counterValue == null ? 0 : Integer.parseInt(counterValue);
}
private void setCounterValue(int i) {
    getSession().getUserData().put(COUNTER, Integer.toString(i));
}
private static final String COUNTER = "jbo.counter";

```

36.7.1.1 What Happens When You Passivate Custom Information

In [Example 36–7](#), when `activateState()` is overridden, the following steps are performed:

1. Search the element for any `jbo.counter` elements.
2. If any are found, loop over the nodes found in the node list.
3. Get first child node of the `jbo.counter` element.

It should be a DOM Text node whose value is the string you saved when your `passivateState()` method above got called, representing the value of the `jbo.counter` attribute.

4. Set the counter value to the activated value from the snapshot.

When `passivateState()` is overridden, it performs the reverse job by doing the following:

1. Retrieve the value of the value to save.
2. Create an XML element to contain the value.
3. Create an XML text node to represent the value.
4. Append the text node as a child of the element.
5. Append the element to the parent element passed in.

Note: The API's used to manipulate nodes in an XML document are provided by the Document Object Model (DOM) interfaces in the `org.w3c.dom` package. These are part of the Java API for XML Processing (JAXP). See the JavaDoc for the `Node`, `Element`, `Text`, `Document`, and `NodeList` interfaces in this package for more details.

36.8 Managing the State of View Objects

By default, all view objects are marked as passivation-enabled, so their state will be saved. However, view objects that have transient attributes do not have those attributes passivated by default. You can change how a view object is passivated, and even which attributes are passivated, using the Tuning page of the view object editor.

36.8.1 How to Manage the State of View Objects

Each view object can be declaratively configured to be passivation-enabled or not. If a view object is not passivation enabled, then no information about it gets written in the application module passivation snapshot.

Performance Tip: There is no need to passivate read-only view objects since they are not intended to be updated and are easily recreated from the XML definition. This eliminates the performance overhead associated with passivation and activation and reduces the CPU usage needed to maintain the application module pool.

To set the passivation state of a view object:

1. In the Application Navigator, double-click a view object to open it in the overview editor.
2. On the General page, expand the **Tuning** section.
3. Select **Passivate State** to make sure the view object data is saved.

Optionally, you can select **Including All Transient Attributes** to passivate all transient attributes at this time, but see [Section 36.8.4, "What You May Need to Know About Passivating Transient View Objects"](#) for additional information.

36.8.2 What You May Need to Know About Passivating View Objects

The activation mechanism is designed to return your view object to the state it was in when the last passivation occurred. To ensure that, Oracle ADF stores in the state snapshot the values of any bind variables that were used for the last query execution. These bind variables are in addition to those that are set on the rowset at the time of passivation. This approach works for an application that does not dynamically reset the WHERE clause and bind variables on each request. In this case, the set of bind variable values used for the last `executeQuery()` and the set of bind variables current on the rowset at passivation time are the same. The passivated state also stores the user-supplied WHERE clause on the view object related to the rowset at the time of passivation.

However, when your application needs to dynamically change WHERE clauses and corresponding bind variables during the span of a single HTTP request, you need to ensure that the user-defined WHERE clause on the view object at the time of passivation matches the set of bind variable values used the last time the view object was executed before passivation. In this case, this is the correct sequence to follow:

1. (Request begins and AM is acquired.)
2. Call `setWhereClause()` on a view object instance that references N bind variables.
3. Call `setWhereClauseParam()` to set the N values for those N bind variables.
4. Call `executeQuery()`.
5. Call `setWhereClause(null)` to clear WHERE clause.

6. Call `setWhereClauseParams(null)` to clear the WHERE clause bind variables.
7. (AM is released.)

If you do not adhere to this strategy of changing runtime view object settings before using them, your application will fail with an SQL exception during application module pooling state activation:

```
JBO-27122: SQLStmtException: <... SQL Statement ...>
```

Because many of the view object's instance settings are saved in the passivation state snapshot and restored on activation (as described in [Section 36.5.1, "What State is Saved?"](#)), it is not advisable to change any of these settings just after executing the view object if you won't be re-executing the view object again during the same block of code (and so, during the same HTTP request). Instead, change the view object instance settings the next time you need them to be different before executing the query.

If you are dynamically adding named WHERE clause parameters to your view object instances, you might find it useful to add the following helper method to your `ViewObjectImpl` framework extension class. This method removes named bind variables that have been added to the view instance at runtime, without removing the ones that have been declaratively defined on the view definition at design time.

```
protected void clearWhereState() {
    ViewDefImpl viewDef = getViewDef();
    Variable[] viewInstanceVars = null;
    VariableManager viewInstanceVarMgr = ensureVariableManager();
    if (viewInstanceVarMgr != null) {
        viewInstanceVars = viewInstanceVarMgr.getVariablesOfKind(Variable.VAR_KIND_
WHERE_CLAUSE_PARAM);
        if (viewInstanceVars != null) {
            for (Variable v: viewInstanceVars) {
                // only remove the variable if its not on the view def.
                if (!hasViewDefVariableNamed(v.getName())) {
                    removeNamedWhereClauseParam(v.getName());
                }
            }
        }
        getDefaultRowSet().setExecuteParameters(null, null, true);
        setWhereClause(null);
        getDefaultRowSet().setWhereClauseParams(null);
    }
}

private boolean hasViewDefVariableNamed(String name) {
    boolean ret = false;
    VariableManager viewDefVarMgr = getViewDef().ensureVariableManager();
    if (viewDefVarMgr != null) {
        try {
            ret = viewDefVarMgr.findVariable(name) != null;
        }
        catch (NoDefException ex) {
            // ignore
        }
    }
    return ret;
}
```

36.8.3 How to Manage the State of Transient View Objects and Attributes

Because view objects are marked as passivated by default, a transient view object — one that contains only transient attributes — is marked to be passivation enabled, but only passivates its information related to the current row and other non-transactional state.

Performance Tip: Transient view object attributes are not passivated by default. Due to their nature, they are usually intended to be read-only and are easily recreated. So, it often doesn't make sense to passivate their values as part of the XML snapshot. This also avoids the performance overhead associated with passivation and activation and reduces the CPU usage needed to maintain the application module pool.

To individually set the passivation state for transient view object attributes:

1. In the Application Navigator, double-click a view object to open it in the overview editor.
2. On the Attributes page, select the transient attribute you want to passivate and click the **Edit** icon.
3. In the Edit Attribute dialog, click the **View Attribute** node.
4. Select the **Passivate** checkbox and click **OK**.

36.8.4 What You May Need to Know About Passivating Transient View Objects

Passivating transient view object attributes is more costly resource-wise and performance-wise, because transactional functionality is usually managed on the entity object level. Since transient view objects are not based on an entity object, this means that all updates are managed in the view object row cache and not in the entity cache. Therefore, passivating transient view objects or transient view object attributes requires special runtime handling.

Usually passivation only saves the values that have been changed, but with transient view objects passivation has to save entire row. The row will include only the view object attributes marked for passivation.

36.8.5 How to Use Transient View Objects to Store Session-level Global Variables

Using passivation, you can use a view object to store one or more global variables, each on a different transient attribute. When you mark a transient attribute as passivated, the ADF Business Components framework will remember the transient values across passivation and activation in high-throughput and failover scenarios. Therefore, it is an easy way to implement a session-level global value that is backed up by the state management mechanism, instead of the less-efficient HTTP Session replication. This also makes it easy to bind to controls in the UI if necessary.

There are two basic approaches to store values between invocations of different screens, one is controller-centric, and the other is model-centric.

Implementation of the task in the ADF controller

The controller-centric approach involves storing and referencing values using attributes in the pageFlowScope. This approach might be appropriate if the global values do not need to be referenced internally by any implementations of ADF Business Components.

For more information about pageFlow scope, see [Section 13.2.10, "What You May Need to Know About Memory Scopes"](#).

Implementation of the task in the ADF model

The model-centric approach involves creating a transient view object, which is conceptually equivalent to a non-database block in Oracle Forms.

1. Create a new view object using the View Object Wizard, as described in [Section 5.2.1, "How to Create an Entity-Based View Object"](#).
 - On step 1 of the wizard, select the option for **Rows populated programmatically, not based on a query**.
 - On step 2, click **New** to define the transient attribute names and types the view object should contain. Make sure to select the **Updateable** option to **Always**.
 - Click **Finish** and the newly-created view object appears in the view object overview editor.
2. Disable any queries from being performed in the view object.
 - On the General page of the overview editor, expand the **Tuning** section, and in the **Retrieve from Database** group box, select the **No Rows** option.
3. Make sure data in the view object is not cleared out during a rollback operation. To implement this, you enable a custom Java class for the view object and override two rollback methods.
 - In the editor, select the **Java** page and click the Edit icon in the **Java Classes** section to open the Java dialog.
 - In the Java dialog, select **Generate View Object Class** and click **OK**.
 - In the overview editor, click on the hyperlink next to the View Object Class in the Java Classes section to open the source editor.
 - From the JDeveloper main menu, click **Source** and then choose **Override Methods**.
 - In the Override Methods dialog, select the `beforeRollback()` and `afterRollback()` methods to override, and click then **OK**.
 - In both the `beforeRollback()` and `afterRollback()` methods, comment out the call to `super` in the Java code.
4. Add an instance of the transient view object to your application module's data model, as described in [Section 9.2.3, "How to Add a View Object to an Application Module"](#).
5. Create an empty row in the view object when a new user begins using the application module.
 - Enable a Java class for your application module if you don't have one yet.
 - Override the `prepareSession()` method of the application module, as described in [Section 9.11.1, "How to Override a Built-in Framework Method"](#).
 - After the call to `super.prepareSession()`, add code to create a new row in the transient view object and insert it into the view object.

Now you can bind read-only and updateable UI elements to the "global" view object attributes just as with any other view object using the data control palette.

36.9 Using State Management for Middle-Tier Savepoints

In the database server you are likely familiar with the savepoint feature that allows a developer to rollback to a certain point within a transaction instead of rolling back the entire transaction. An application module offers the same feature but implemented in the middle tier.

36.9.1 How to Use State Management for Savepoints

To use state management for implementing middle-tier savepoints, you override three methods in the `oracle.jbo.ApplicationModule` interface

```
public String passivateStateForUndo(String id, byte[] clientData, int flags)
public byte[] activateStateForUndo(String id, int flags)
public boolean isValidIdForUndo(String id)
```

You can use these methods to create a stack of named snapshots and restore the pending transaction state from them by name. Keep in mind that those snapshots do not survive past duration of transaction (for example, events of commit or rollback). This feature could be used to develop complex capabilities of the application, such as the ability to undo and redo changes. Another ambitious goal that could exploit this functionality would be functionality to make the browser back and forward buttons behave in an application-specific way. Otherwise, simple uses of these methods can come quite in handy.

36.10 Testing to Ensure Your Application Module is Activation-Safe

If you have not *explicitly* tested that your application module functions when its pending state gets activated from a passivation snapshot, then you may encounter an unpleasant surprise in your production environment when heavy system load tests this aspect of your system for the first time.

36.10.1 Understanding the `jbo.ampool.doampooling` Configuration Parameter

The `jbo.ampool.doampooling` configuration property corresponds to the **Enable Application Module Pooling** option in the **Pooling and Scalability** tab of the Configuration Editor. By default, this checkbox is checked so that application module pooling is enabled. Whenever you deploy your application in a production environment the default setting of `jbo.ampool.doampooling` to `true` is the way you will run your application. But, as long as you run your application in a test environment, setting the property to `false` can play an important role in your testing. When this property is `false`, there is effectively no application pool. When the application module instance is released at the end of a request it is immediately removed. On subsequent requests made by the same user session, a new application module instance must be created to handle it and the pending state of the application module must be reactivated from the passivation store.

36.10.2 Disabling Application Module Pooling to Test Activation

As part of your overall testing plan, you should adopt the practice of testing your application modules with the `jbo.ampool.doampooling` configuration parameter set to `false`. This setting completely disables application module pooling and forces the system to activate your application module's pending state from a passivation snapshot on *each* page request. It is an excellent way to detect problems that might occur in your production environment due to assumptions made in your custom application code.

Caution: It is important to reenable application module pooling after you conclude testing and are ready to deploy the application to a production environment. The configuration property `jbo.ampool.doampooling` set to `false` is not a supported configuration for production applications and must be set to `true` before deploying the application.

For example, if you have transient view object attributes you believe *should* be getting passivated, this technique allows you to test that they are working as you expect. In addition, consider situations where you might have introduced:

- Private member fields in application modules, view objects, or entity objects
- Custom user session state in the `Session` user data hashtable

Your custom code likely assumes that this custom state will be maintained across HTTP requests. As long as you test with a single user on the JDeveloper Integrated WLS server, or test with a small number of users, things will appear to work fine. This is due to the "stateless with affinity" optimization of the ADF application module pool. If system load allows, the pool will continue to return the same application module instance to a user on subsequent requests. However, under heavier load, during real-world use, it may not be able to achieve this optimization and will need to resort to grabbing any available application module instance and reactivating its pending state from a passivation snapshot. If you have not correctly overridden `passivateState()` and `activateState()` (as described in [Section 36.7, "Managing Custom User-Specific Information"](#)) to save and reload your custom component state to the passivation snapshot, then your custom state will be missing (i.e. `null` or back to your default values) after this reactivation step. Testing with `jbo.ampool.doampooling` set to `false` allows you to quickly isolate these kinds of situations in your code.

36.11 Keeping Pending Changes in the Middle Tier

The ADF state management mechanism relies on passivation and activation to manage the state of an application module instance. Implementing this feature in a robust way is only possible if all pending changes are managed by the application module transaction in the middle tier. The most scalable strategy is to keep pending changes in middle-tier objects and not perform operations that cause pending database state to exist across HTTP requests. This allows the highest leverage of the performance optimizations offered by the application module pool and the most robust runtime behavior for your application.

When the `jbo.doconnectionpooling` configuration parameter is set to `true` — typically in order to share a common pool of database connections across multiple application module pools — upon releasing your application module to the application module pool, its JDBC connection is released back to the database connection pool and a `ROLLBACK` will be issued on that connection. This implies that all changes which were posted but *not committed* will be lost. On the next request, when the application module is used, it will receive a JDBC connection from the pool, which may be a different JDBC connection instance from the one it used previously. Those changes that were posted to the database but not committed during the previous request are lost.

Caution: When the `jbo.doconnectionpooling` configuration parameter is set to `true` — typically in order to share a common pool of database connections across multiple application module pools — upon releasing your application module to the application module pool, its JDBC connection is released back to the database connection pool and a `ROLLBACK` will be issued on that connection. This implies that all changes which were posted but *not committed* will be lost. On the next request, when the application module is used, it will receive a JDBC connection from the pool, which may be a different JDBC connection instance from the one it used previously. Those changes that were posted to the database but not committed during the previous request are lost.

The `jbo.doconnectionpooling` configuration parameter is set by checking the **Disconnect Application Module Upon Release** property on the **Pooling and Scalability** tab of the Configuration Editor.

36.11.1 How to Set Applications to Use Optimistic Locking

Oracle recommends using optimistic locking for web applications. Pessimistic locking, which is the default, should not be used for web applications as it creates pending transactional state in the database in the form of row-level locks. If pessimistic locking is set, state management will work, but the locking mode will not perform as expected. Behind the scenes, every time an application module is recycled, a rollback is issued in the JDBC connection. This releases all the locks that pessimistic locking had created.

Performance Tip: Oracle recommends using optimistic locking for web applications. Only optimistic locking is compatible with the application module unmanaged release level mode, which allows the application module instance to be immediately released when a web page terminates. This provides the best level of performance for web applications that expect many users to access the application simultaneously.

To change your configuration to use optimistic locking, open the **Properties** tab of the Configuration Editor and set the value of the `jbo.locking.mode` to `optimistic` or `optupdate`.

Optimistic locking (`optimistic`) issues a `SELECT FOR UPDATE` statement to lock the row, then detects whether the row has been changed by another user by comparing the change indicator attribute — or, if no change indicator is specified, the values of all the persistent attributes of the current entity as they existed when the entity object was fetched into the cache.

Optimistic update locking (`optupdate`) does not perform any locking. The `UPDATE` statement determines whether the row was updated by another user by including a `WHERE` clause that will match the existing row to update only if the attribute values are unchanged since the current entity object was fetched.

36.11.2 How to Avoid Clashes Using the `postChanges()` Method

The transaction-level `postChanges()` method exists to force the transaction to post unvalidated changes without committing them. This method is not recommended for use in web applications unless you can guarantee that the transaction will definitely be committed or rolled-back during the same HTTP request. Failure to heed this advice

can lead to strange results in an environment where both application modules and database connections can be pooled and shared serially by multiple different clients.

36.11.3 How to Use the Reserved Level For Pending Database States

If for some reason you need to create a transactional state in the database in some request by invoking `postChanges()` method or by calling PL/SQL stored procedure, but you cannot issue a commit or rollback by the end of that same request, then you *must* release the application module instance with the **reserved** level from that request until a subsequent request when you either commit or rollback.

Performance Tip: Oracle recommends as short a period of time as possible between creation of transactional state in the database performing the concluding commit or rollback. This will ensure that reserved level doesn't have to be used for a long time, as it has adverse effects on application's scalability and reliability.

Once an application module has been released with reserved level, it remains at that release level for all subsequent requests until release level is explicitly changed back to managed or unmanaged level. So, it is your responsibility to set release level back to managed level once commit or rollback has been issued.

For more information, see [Section 36.4, "Setting the Application Module Release Level at Runtime"](#).

Understanding Application Module Pooling

This chapter describes how ADF Business Components application module pools work and how you can tune the pools to optimize application performance.

This chapter includes the following sections:

- [Section 37.1, "Overview of Application Module Pooling"](#)
- [Section 37.2, "Understanding Configuration Property Scopes"](#)
- [Section 37.3, "Setting Pool Configuration Parameters"](#)
- [Section 37.4, "How Many Pools are Created, and When?"](#)
- [Section 37.5, "Application Module Pool Parameters"](#)
- [Section 37.6, "Database Connection Pool Parameters"](#)
- [Section 37.7, "How Database and Application Module Pools Cooperate"](#)
- [Section 37.8, "Database User State and Pooling Considerations"](#)

37.1 Overview of Application Module Pooling

An application module pool is a collection application module instances of the same type. For example, the Fusion Order Demo application has one or more instances of the `StoreServiceAM` application module in it, based on the number of users that are visiting the site. This pool of application module instances is shared by multiple browser clients whose typical "think time" between submitting web pages allows optimizing the number of application module components to be effectively smaller than the total number of active users working on the system. That is, twenty users visiting the web site from their browser might be able to be serviced by 5 or 10 application module instances instead of having as many application module instances as you have browser users.

Application module components can be used to support Fusion web application scenarios that are completely stateless, or they can be used to support a unit of work that spans multiple browser pages. As a performance optimization, when an instance of an application module is returned to the pool in "managed state" mode, the pool keeps track that the application module is referenced by that particular session. The application module instance is still in the pool and available for use, but it would *prefer* to be used by the same session that was using it last time because maintaining this so-called "session affinity" improves performance.

So, at any one moment in time, the instances of application modules in the pool are logically partitioned into three groups, reflecting their state:

- Unconditionally *available* for use

- Available for use, but *referenced* for session affinity reuse by an active user session
- *Unavailable*, inasmuch as it's currently in use (at that very moment) by some thread in the web container.

[Section 37.2, "Understanding Configuration Property Scopes"](#) describes the application module pool configuration parameters and how they affect the behavior of the pool.

37.2 Understanding Configuration Property Scopes

Each runtime configuration property used by ADF Business Components has a scope. The scope of each property indicates at what level the property's value is evaluated and whether its value is effectively shared (i.e. static) in a single Java VM, or not. The ADF Business Components `PropertyManager` class is the registry of all supported properties. It defines the property names, their default values, and their scope. This class contains a `main()` method so that you can run the class from the command line to see a list of all the configuration property information.

Assuming `JDEVHOME` is the JDeveloper installation directory, to see this list of settings for reference, do the following:

```
$ java -cp JDEVHOME/BC4J/lib/bc4jmt.jar oracle.jbo.common.PropertyManager
```

Issuing this command will send all of the ADF Business Components configuration properties to the console. It also lists a handy reference about the different levels at which you can set configuration property values and remind you of the precedence order these levels have:

```
-----
Properties loaded from following sources, in order:
1. Client environment [Provided programmatically
   or declaratively in bc4j.xcfg]
2. Applet tags
3. -D flags (appear in System.properties)
4. bc4j.properties file (in current directory)
5. /oracle/jbo/BC4J.properties resource
6. /oracle/jbo/common.jboserver.properties resource
7. /oracle/jbo/common.Diagnostic.properties resource
8. System defined default
-----
```

You'll see each property is listed with one of the following scopes:

- `MetaObjectManager`

Properties at this scope are initialized once per Java VM when the ADF `PropertyManager` is first initialized.

- `SessionImpl`

Properties at this scope are initialized once per invocation of `ApplicationModule.prepareSession()`.

- `Configuration`

Properties at this scope are initialized when the `ApplicationModule` pool is first created and the application module's configuration is read the first time.

- `Diagnostic`

Properties at this scope are specific to the built-in ADF Business Components diagnostic facility.

At each of these scopes, the layered value resolution described above is performed when the properties are initialized. Whenever property values are initialized, if you have specified them in the *Client Environment* (level 1 in the resolution order) the values will take precedence over values specified as System parameters (level 3 in the resolution order).

The Client Environment is a hashtable of name/value pairs that you can either programmatically populate, or which will be automatically populated for you by the Configuration object when loaded, with the name/value pairs it contains in its entry in the `bc4j.xcfg` file. The implication of this is that for any properties scoped at MetaObjectManager level, the most reliable way to ensure that all of your application modules use the same default value for those properties is to do both of the following:

1. Make sure the property value does not appear in any of your application module's `bc4j.xcfg` file configuration name/value pair entries.
2. Set the property value using a Java system property in your runtime environment.

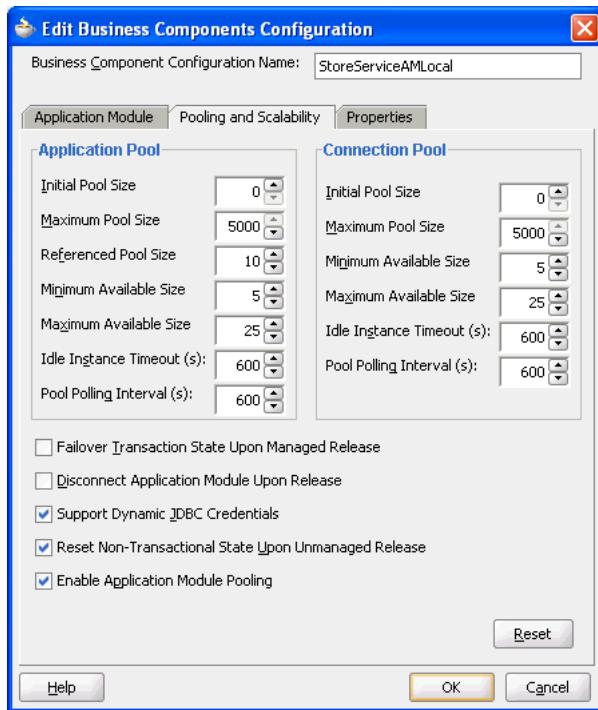
If, instead, you leave any MetaObjectManager-scoped properties in your `bc4j.xcfg` files, you will have the undesirable behavior that they will take on the value specified in the configuration of the *first* application module whose pool gets created after the Java VM starts up.

37.3 Setting Pool Configuration Parameters

You control the runtime behavior of an application module pool by setting appropriate configuration parameters. You can set these declaratively in an application module configuration, supply them as Java System parameters, or set them programmatically at runtime.

37.3.1 Setting Configuration Properties Declaratively

The **Pooling and Scalability** tab of the Configuration Editor shown in [Figure 37-1](#) is used for seeing and setting parameters.

Figure 37–1 Pooling and Scalability Tab of the Configuration Manager

The values that you supply through the **Configuration Manager** are saved in an XML file named `bc4j.xcfg` in the `./common` subdirectory relative to the application module's XML component definition. All of the configurations for all of the application modules in a single Java package are saved in that same file.

For example, if you look at the `bc4j.xcfg` file in the `.src/oracle/fodemo/storefront/store/service/common` directory of the Fusion Order Demo application's StoreFront project, you will see the three named configurations for its `StoreServiceAM` application module, as shown in [Example 37–1](#). In this case, The `StoreServiceAMLocal` and the `StoreServiceAMShared` configurations uses JDBC URL connections for use by the Business Component Browser. The `StoreFrontService` uses JDBC Datasource names and is used by the Fusion web application. The connection details for each JDBC connection appear in the `connections.xml` file located in the `./.adf/META-INF` subdirectory relative to the project directory.

Example 37–1 Configuration Settings for the SRService Application Module

```
<BC4JConfig version="11.0" xmlns="http://xmlns.oracle.com/bc4j/configuration">
  < AppModuleConfigBag>
    < AppModuleConfig name="StoreServiceAMLocal"
      ApplicationName="oracle.fodemo.storefront.store.service.StoreServiceAM"
      DeployPlatform="LOCAL"
      JDBCName="FOD"
      jbo.project="StoreFrontService"
      java.naming.factory.initial="oracle.jbo.common.JboInitialContextFactory">
      < Database jbo.locking.mode="optimistic"/>
      < Security AppModuleJndiName="oracle.fodemo.storefront.store.service.StoreServiceAM"/>
    </ AppModuleConfig>
    < AppModuleConfig name="StoreServiceAMShared"
      ApplicationName="oracle.fodemo.storefront.store.service.StoreServiceAM"
      DeployPlatform="LOCAL"
      JDBCName="FOD"
```

```

jbo.project="StoreFrontService"
java.naming.factory.initial="oracle.jbo.common.JboInitialContextFactory">
<AM-Pooling jbo.ampool.isuseexclusive="false" jbo.ampool.maxpoolsize="1"/>
<Security AppModuleJndiName="oracle.fodemo.storefront.store.service.StoreServiceAM"/>
</ AppModuleConfig>
< AppModuleConfig name="StoreFrontService"
    ApplicationName="oracle.fodemo.storefront.store.service.StoreServiceAM"
    DeployPlatform="LOCAL"
    JDBCName="FOD"
    jbo.project="StoreFrontService"
    java.naming.factory.initial="oracle.jbo.common.JboInitialContextFactory">
<AM-Pooling jbo.ampool.resetnontransactionalstate="false"/>
<Database jbo.locking.mode="optimistic"/>
<Security AppModuleJndiName="oracle.fodemo.storefront.store.service.StoreServiceAM"/>
<Custom JDBCDataSource="jdbc/FODDS"/>
</ AppModuleConfig>
</ AppModuleConfigBag>
</BC4JConfig>
```

Note that attributes of the `<AppModuleConfig>` tag appear with names matching their ADF Business Component properties (for example, the attribute `jbo.locking.mode` corresponds to the property `jbo.locking.mode`). It's also important to understand that if a property is currently set to its runtime *default* value, then the **Configuration Manager** does *not* write the entry to the `bc4j.xcfg` file.

37.3.2 Setting Configuration Properties as System Parameters

As an alternative to specifying configuration properties in the `bc4j.xcfg` file, you can also set Java VM system parameters with the same property names. These system parameters will be used *only if* a corresponding property *does not exist* in the relevant `bc4j.xcfg` file for the application module in question. In other words, configuration parameters that appear in the application module configuration take precedence over parameters of the same name supplied as Java system parameters.

You typically set Java system parameters using the `-D` command line flag to the Java VM like this:

```
java -Dproperty=value -jar yourserver.jar
```

Alternatively, your Java EE container probably has a section in its own configuration files where Java system parameters can be specified for use at Java EE container startup time.

If you adopt the technique of specifying site-specific *default* values for Oracle ADF configuration parameters as Java system parameters, you should make sure that your application's `bc4j.xcfg` files do *not* include references to these parameters unless you want to define an application-module-specific exception to these global default values.

Caution: The values of **Idle Instance Timeout**, **Pool Polling Interval** settings for both the **Application Pool** and the database **Connection Pool** are displayed and edited in this dialog as a number of *seconds*, but are saved to the configuration file in milliseconds. If you provide a value for any of these four parameters as a Java System parameter — or if you hand-edit the `bc4j.xcfg` file — make sure to provide these time interval values in milliseconds!

37.3.3 Programmatically Setting Configuration Properties

You can set configuration properties programmatically by creating a Java class that implements the `EnvInfoProvider` interface in the `oracle.jbo.common.ampool` package. In your class, you override the `getInfo()` method and call `put()` to put values into the environment Hashtable passed in as shown in [Example 37–2](#).

Example 37–2 Setting Environment Properties with a Custom EnvInfoProvider

```
package devguide.advanced.customenv.view;
import java.util.Hashtable;
import oracle.jbo.common.ampool.EnvInfoProvider;
/**
 * Custom EnvInfoProvider implementation to set
 * environment properties programmatically
 */
public class CustomEnvInfoProvider implements EnvInfoProvider {
    /**
     * Overridden framework method to set custom values in the
     * environment hashtable.
     *
     * @param string - ignore
     * @param environment Hashtable of config parameters
     * @return null - not used
     */
    public Object getInfo(String string, Object environment) {
        Hashtable envHashtable = (Hashtable)environment;
        envHashtable.put("some.property.name", "some value");
        return null;
    }
    /* Required to implement EnvInfoProvider */
    public void modifyInitialContext(Object object) {}
    /* Required to implement EnvInfoProvider */
    public int getNumOfRetries() {return 0;}
}
```

When creating an application module for a stateless or command-line-client, with the `createRootApplicationModule()` method of the `Configuration` class, you can pass the custom `EnvInfoProvider` as the optional second argument. In order to use a custom `EnvInfoProvider` in an ADF web-based application, you need to implement a custom session cookie factory class as shown in [Example 37–3](#). To use your custom session cookie factory, set the `jbo.ampool.sessioncookiefactoryclass` configuration property to the fully-qualified name of your custom session cookie factory class.

Example 37–3 Custom SessionCookieFactory to Install a Custom EnvInfoProvider

```
package devguide.advanced.customenv.view;
import java.util.Properties;
import oracle.jbo.common.ampool.ApplicationPool;
import oracle.jbo.common.ampool.EnvInfoProvider;
import oracle.jbo.common.ampool.SessionCookie;
import oracle.jbo.http.HttpSessionCookieFactory;
/**
 * Example of custom http session cookie factory
 * to install a custom EnvInfoProvider implementation
 * for an ADF web-based application.
 */
public class CustomHttpSessionCookieFactory
    extends HttpSessionCookieFactory {
```

```

public SessionCookie createSessionCookie(String appId,
                                         String sessionId,
                                         ApplicationPool pool,
                                         Properties props) {
    SessionCookie cookie =
        super.createSessionCookie(appId, sessionId, pool, props);
    EnvInfoProvider envInfoProv = new CustomEnvInfoProvider();
    cookie.setEnvInfoProvider(envInfoProv);
    return cookie;
}
}

```

37.4 How Many Pools are Created, and When?

There are two kinds of pools in use when running a typical Fusion web application, Application Module pools and database connection pools. It's important to understand how many of each kind of pool your application will create.

37.4.1 Application Module Pools

Application Module components can be used at runtime in two ways:

- As an application module the client accesses directly
- As a reusable component aggregated (or "nested") inside of another application module instance

When a client accesses it directly, an application module is called a *root application module*. Clients access nested application modules *indirectly* as a part of their containing application module instance. It's possible, but not common, to use the same application module at runtime in both ways. The important point is that ADF only creates an application module pool for a *root* application module.

The basic rule is that one application module pool is created for each *root* application module used by a Fusion web application in *each* Java VM where a root application module of that type is used by the ADF controller layer.

37.4.2 Database Connection Pools

Fusion web applications always use a database connection pool, but which one they use for your application modules depends on how they define their connection:

- JDBC URL (e.g. `jdbc:oracle:thin:@penguin:1521:ORCL`)
- JNDI Name for a Datasource (e.g. `java:comp/env/jdbc/YourConnectionDS`)

If you supply a JDBC URL connection while configuring your application module — which happens when you select a JDeveloper named connection which encapsulates the JDBC URL and username information — then the ADF database connection pool will be used for managing the connection pool.

If you supply the JNDI name of a JDBC Datasource then the ADF database connection pool will *not* be used and the configuration parameters described below relating to the ADF database connection pool are not relevant.

Note: To configure the database connection pool for JDBC Datasources looked-up by JNDI from your Java EE web and/or EJB container, consult the documentation for your Java EE container to understand the pooling configuration options and how to set them.

When using ADF database connection pooling, you have the following basic rule: One database connection pool is created for each unique <JDBCURL,Username> pair, in *each* Java VM where a <JDBCURL,Username> connection is requested by a root application used by the ADF controller layer.

37.4.3 Understanding Application Module and Connection Pools

The number of pools and the type of pools that your application will utilize will depend upon how the target platform is configured. For example, will there be more than one Java Virtual Machine (JVM) available to service the web requests coming from your application users and will there be more than one Oracle Application Server instance? To understand how many pools of which kinds are created for an application in both a single-JVM scenario and a multiple-JVM runtime scenario, review the following assumptions:

- Your Fusion web application makes use of two application modules `HRModule` and `PayablesModule`.
- You have a `CommonLOVModule` containing a set of commonly used view objects to support list of values in your application, and that both `HRModule` and `PayablesModule` aggregate a nested instance of `CommonLOVModule` to access the common LOV view objects it contains.
- You have configured both `HRModule` and `PayablesModule` to use the same JDeveloper connection definition named `appuser`.
- In both `HRModule` and `PayablesModule` you have configured `jbo.passivationstore=database` (the default) and configured the ADF "internal connection" (`jbo.server.internal_connection`) used for state management persistence to have the value of a fully-qualified JDBC URL that points to a *different* username than the `appuser` connection does.

37.4.3.1 Single Oracle Application Server Instance, Single Oracle WebLogic Server Instance, Single JVM

If you deploy this application to a single Oracle Application Server instance, configured with a single Oracle WebLogic Server instance having a single Java VM, there is only a single Java VM available to service the web requests coming from your application users.

Assuming that all the users are making use of web pages that access both the `HRModule` and the `PayablesModule`, this will give:

- One application module pool for the `HRModule` root application module
- One application module pool for the `PayablesModule` root application module
- One DB connection pool for the `appuser` connection
- One DB connection pool for the JDBC URL supplied for the internal connection for state management.

This gives a total of two application module pools and two database pools in this single Java VM.

Note: There is no separate application module pool for the nested instances of the reusable `CommonLOVModule`. Instances of `CommonLOVModule` are wrapped by instances of `HRModule` and `PayablesModule` in their respective application module pools.

37.4.3.2 Multiple Oracle Application Server Instances, Single Oracle WebLogic Server Instance, Multiple JVMs

Next consider a deployment environment involving multiple Java VMs. Assume that you have installed Oracle Application Server onto two different physical machines, with a hardware load-balancer in front of it. On each of these two machines, imagine that the Oracle Application Server instance is configured to have one Oracle WebLogic Server instance with two JVMs. As users of your application access the application, their requests are shared across these two Oracle Application Server instances, and within each Oracle Application Server instance, across the two JVMs that its Oracle WebLogic Server instance has available.

Again assuming that all the users are making use of web pages that access both the HRModule and the PayablesModule, this will give:

- Four application module pools for HRModule, one in each of four JVMs.
 $(1 \text{ HRModule root application module}) \times (2 \text{ Oracle Application Server Instances}) \times (2 \text{ Oracle WebLogic Server JVMs each})$
- Four application module pools for PayablesModule, one in each of four JVMs.
 $(1 \text{ PayablesModule root application module}) \times (2 \text{ Oracle Application Server Instances}) \times (2 \text{ Oracle WebLogic Server JVMs each})$
- Four DB connection pools for appuser, one in each of four JVMs.
 $(1 \text{ appuser DB connection pool}) \times (2 \text{ Oracle Application Server Instances}) \times (2 \text{ Oracle WebLogic Server JVMs each})$
- Four DB connection pools for the internal connection JDBC URL, one in each of four JVMs.
 $(1 \text{ internal connection JDBC URL DB connection pool}) \times (2 \text{ Oracle Application Server Instances}) \times (2 \text{ Oracle WebLogic Server JVMs each})$

This gives a total of eight application module pools and eight DB connection pools spread across four JVMs.

As you begin to explore the configuration parameters for the application module pools in [Section 37.5, "Application Module Pool Parameters"](#), keep in mind that the parameters apply to a given application module pool for a given application module in a single JVM.

As the load balancing spreads user request across the multiple JVMs where ADF is running, each individual application module pool in each JVM will have to support one n^{th} of the user load — where N is number of JVMs available to service those user requests. The appropriate values of the application module and DB connection pools need to be set with the number of Java VMs in mind. The basic approach is to base sizing parameters on load testing and the results of the application module pooling statistics, then divide that total number by the N pools you will have based on your use of multiple application servers and multiple Oracle WebLogic Server instances. For example, if you decide to set the minimum number of application modules in the pool to ten and you end up with five pools due to having five Oracle WebLogic Server instances servicing this application, then you would want to configure the parameter to 2 (ten divided by five), not 10 (which would only serve a given application module in a single JVM).

37.5 Application Module Pool Parameters

The application module pool configuration parameters fall into three logical categories relating to pool behavior, pool sizing, and pool cleanup behavior.

37.5.1 Pool Behavior Parameters

[Table 37–2](#) lists the application module configuration parameters that affect the behavior of the application module pool.

Table 37–1 Application Module Pool Behavior Configuration Parameters

Pool Configuration Parameter	Description
Failover Transaction State Upon Managed Release (jbo.dofailover)	Enables eager passivation of pending transaction state each time an application module is released to the pool in "Managed State" mode. See Section 36.2.2.2, "How Passivation Changes When Optional Failover Mode is Enabled" for more information. Fusion web applications should set enable failover (true) to allow any other application module to activate the state at any time. This feature is disabled by default (false)
Row-Level Locking Behavior Upon Release (jbo.locking.mode)	Forces the application module pool not to create a pending transaction state on the database with row-level locks each time the application module is released to the pool. See Section 36.11.1, "How to Set Applications to Use Optimistic Locking" for more information. Fusion web applications should set the locking mode to optimistic to avoid creating the row-level locks. This feature is disabled by default (pessimistic).
Disconnect Application Module Upon Release (jbo.doconnectionpooling)	Forces the application module pool to release the JDBC connection used each time the application module is released to the pool. See Section 37.7, "How Database and Application Module Pools Cooperate" for more information. This feature is disabled by default (false).
Enable Application Module Pooling (jbo.ampool.dopooling)	Enables application module pooling by default. Whenever you deploy your application in a production environment the default setting of jbo.ampool.dopooling to true and is the way you will run your application. But, as long as you run your application in a test environment, setting the property to false can play an important role in your testing. When this property is false, there is effectively no application pool. See Section 36.10, "Testing to Ensure Your Application Module is Activation-Safe" for more information. This feature is disabled by default (false)

Table 37-1 (Cont.) Application Module Pool Behavior Configuration Parameters

Pool Configuration Parameter	Description
Support Dynamic JDBC Credentials (jbo.ampool.dynamicjdbccredentials)	Enables additional pooling lifecycle events to allow developer-written code to change the database credentials (username/password) each time a new user session begins to use the application module.
Reset Non-Transactional State Upon Unmanaged Release (jbo.ampool.resetnontransactionalstate))	This feature is enabled by default (<code>true</code>), however this setting is a necessary but not sufficient condition to implement the feature. The complete implementation requires additional developer-written code.
Reset Non-Transactional State Upon Unmanaged Release (jbo.ampool.resetnontransactionalstate))	Forces the application module to reset any non-transactional state like view object runtime settings, JDBC prepared statements, bind variable values, etc. when the application module is released to the pool in unmanaged or "stateless" mode.
Reset Non-Transactional State Upon Unmanaged Release (jbo.ampool.resetnontransactionalstate))	This feature is enabled by default (<code>true</code>). Disabling this feature can improve performance, however since it does not clear bind variable values, your application needs to ensure that it systemically sets bind variable values correctly. Failure to do so with this feature disabled can mean one user might see data with another users bind variable values.)

37.5.2 Pool Sizing Parameters

[Table 37-2](#) lists the application module configuration parameters that affect the sizing of the application module pool.

Table 37-2 Application Module Pool Sizing Configuration Parameters

Pool Configuration Parameter	Description
Initial Pool Size (jbo.ampool.initpoolsize)	The number of application module instances to created when the pool is initialized.
	The default is 0 (zero) instances. A general guideline is to configure this to 10% more than the anticipated number of concurrent application module instances required to service all users.
Maximum Pool Size (jbo.ampool.maxpoolsize)	Creating application module instances during initialization takes the CPU processing costs of creating application module instances during the initialization instead of on-demand when additional application module instances are required.
	The maximum number of application module instances that the pool can allocate. The pool will never create more application module instances than this limit imposes.
	The default is 5000 instances. A general guideline is to configure this to 20% more than the initial pool size to allow for some additional growth. If this is set too low, then some users may see an error accessing the application if no application module instances are available.

Table 37–2 (Cont.) Application Module Pool Sizing Configuration Parameters

Pool Configuration Parameter	Description
Referenced Pool Size (jbo.recyclethreshold)	The maximum number of application module instances in the pool that attempt to preserve session affinity for the next request made by the session which used them last before releasing them to the pool in managed-state mode.
	The referenced pool size should always be less than or equal to the maximum pool size. The default is to allow 10 available instances to try and remain "loyal" to the affinity they have with the most recent session that released them in managed state mode.
Maximum Instance Time to Live (jbo.ampool.timetolive)	Configure this value to maintain the application module instance's affinity to a user's session. A general guideline is to configure this to the expected number of concurrent users that perform multiple operations with short think times. If there are no users expected to use the application with short think times, then this can be configured to 0 zero to eliminate affinity.
	Maintaining this affinity as much as possible will save the CPU processing cost of needing to switch an application module instance from one user session to another.
	The number of milliseconds after which to consider an application module instance in the pool as a candidate for removal during the next resource cleanup regardless of whether it would bring the number of instances in the pool below minavailablesize.
	The default is 3600000 milliseconds of total time to live (which is 3600 seconds, or one hour). The default value is sufficient for most applications.

37.5.3 Pool Cleanup Parameters

A single "application module pool monitor" per Java VM runs in a background thread and wakes up every so often to do resource reclamation. [Table 37–3](#) lists the parameters that affect how resources are reclaimed when the pool monitor does one of its resource cleanup passes.

Note: Since there is only a single application monitor pool monitor per Java VM, the value that will effectively be used for the application module pool monitor polling interval will be the value found in the application module configuration read by the *first* application module pool that gets created. To make sure this value is set in a predictable way, it is best practice to set all application modules to use the same **Pool Polling Interval** value.

Table 37-3 Application Module Resource Management Configuration Parameters

Pool Configuration Parameter	Description
Pool Polling Interval (jbo.ampool.monitorsleepinterval)	The length of time in milliseconds between pool resource cleanup.
While the number of application module instances in the pool will never exceed the maximum pool size, available instances which are candidates for getting removed from the pool do not get "cleaned up" until the next time the application module pool monitor wakes up to do its job.	The default is to have the application module pool monitor wake up every 600000 milliseconds (which is 600 seconds, or ten minutes). Configuring a lower interval results in inactive application module instances being removed more frequently to save memory. Configuring a higher interval results in less frequent resource cleanups.
Maximum Available Size (jbo.ampool.maxavailablesize)	The ideal maximum number of available application module instances in the pool when not under abnormal load.
	When the pool monitor wakes up to do resource cleanup, it will try to remove available application module instances to bring the total number of available instances down to this ideal maximum. Instances that have been not been used for a period longer than the idle instance time-out will always get cleaned up at this time, then additional available instances will be removed if necessary to bring the number of available instances down to this size.
The default maximum available size is 25 instances. Configure this to leave the maximum number of available instances desired after a resource cleanup. A lower value generally results in more application module instances being removed from the pool on a cleanup.	
Minimum Available Size (jbo.ampool.minavailablesize)	The minimum number of available application module instances that the pool monitor should leave in the pool during a resource cleanup operation.
Set to 0 (zero) if you want the pool to shrink to contain no instances when all instances have been idle for longer than the idle time-out after a resource cleanup.	
	The default is 5 instances.

Table 37–3 (Cont.) Application Module Resource Management Configuration Parameters

Pool Configuration Parameter	Description
Idle Instance Timeout (jbo.ampool.maxinactiveage)	The number of milliseconds after which to consider an inactive application module instance in the pool as a candidate for removal during the next resource cleanup.
Maximum Instance Time to Live (jbo.ampool.timetolive)	The default is 600000 milliseconds of idle time (which is 600 seconds, or ten minutes). A lower value results in more application module instances being marked as a candidate for removal at the next resource cleanup. A higher value results in fewer application module instances being marked as a candidate for removal at the next resource cleanup.
	The number of milliseconds after which to consider a connection instance in the pool as a candidate for removal during the next resource cleanup regardless of whether it would bring the number of instances in the pool below minavailablesize.

The default is 3600000 milliseconds of total time to live (which is 3600 seconds, or one hour). A lower value reduces the time an application module instance can exist before it must be removed at the next resource cleanup. The default value is sufficient for most applications. A higher value increases the time an application module instance can exist before it must be removed at the next cleanup.

37.6 Database Connection Pool Parameters

If you are using a JDBC URL for your connection information so that the ADF database connection pool is used, then configuration parameters listed in [Table 37–4](#) can be used to tune the behavior of the database connection pool. A single "database connection pool monitor" per Java VM runs in a background thread and wakes up every so often to do resource reclamation. The parameters in [Table 37–3](#) include the ones that affect how resources are reclaimed when the pool monitor does one of its resource cleanup passes.

Note: The configuration parameters for database connection pooling have MetaObjectManager scope (described in [Section 37.2, "Understanding Configuration Property Scopes"](#) earlier). This means their settings are global and will be set once when the first application module pool in your application is created. To insure the most predictable behavior, Oracle recommends leaving the values of these parameters in the **Connection Pool** section of the **Pooling and Scalability** tab at their *default* values — so that no entry for them is written into the `bc4j.xcfg` file — and to instead set the desired values for the database connection pooling tuning parameters as Java System Parameters in your Java EE container.

Table 37–4 Database Connection Pool Parameters

Pool Configuration Parameter	Description
Initial Pool Size (jbo.initpoolsize)	The number of JDBC connection instances to be created when the pool is initialized. The default is an initial size of 0 instances.
Maximum Pool Size (jbo.maxpoolsize)	The maximum number of JDBC connection instances that the pool can allocate. The pool will never create more JDBC connections than this imposes. The default is 5000 instances.
Pool Polling Interval (jbo.poolmonitorsleepinterval)	The length of time in milliseconds between pool resource cleanup.
	While the number of JDBC connection instances in the pool will never exceed the maximum pool size, available instances which are candidates for getting removed from the pool do not get "cleaned up" until the next time the JDBC connection pool monitor wakes up to do its job.
Maximum Available Size (jbo.poolmaxavailablesize)	The default is 600000 milliseconds of idle time (which is 600 seconds, or ten minutes). Configuring a lower interval results in inactive connection instances being removed more frequently to save memory. Configuring a higher interval results in less frequent resource cleanups.
	The ideal maximum number of JDBC connection instances in the pool when not under abnormal load. When the pool monitor wakes up to do resource cleanup, it will try to remove available JDBC connection instances to bring the total number of available instances down to this ideal maximum. Instances that have been not been used for a period longer than the idle instance time-out will always get cleaned up at this time, then additional available instances will be removed if necessary to bring the number of available instances down to this size. The default is an ideal maximum of 25 instances (when not under load).
Minimum Available Size (jbo.poolminavailablesize)	The minimum number of available JDBC connection instances that the pool monitor should leave in the pool during a resource cleanup operation. Set to zero (0) if you want the pool to shrink to contain no instances when all instances have been idle for longer than the idle time-out.
Idle Instance Timeout (jbo.poolmaxinactiveage)	The default is to not let the minimum available size drop below 5 instances.
	The number of seconds after which to consider an inactive JDBC connection instance in the pool as a candidate for removal during the next resource cleanup.
	The default is 600000 milliseconds of idle time (which is 600 seconds, or ten minutes).

Notice that since the database connection pool does not implement the heuristic of session affinity, there is *no* configuration parameter for the database connection pool which controls the referenced pool size.

37.7 How Database and Application Module Pools Cooperate

How ADF application module pools use the database connection pool depends on the setting of the `jbo.doconnectionpooling` application module configuration parameter. In the **Configuration Manager** panel that you see in [Figure 37-1](#), you set this parameter using the checkbox labelled **Disconnect Application Module Upon Release**.

Note: The notion of *disconnecting* the application module upon release to the pool better captures what the actual feature is doing than the related configuration parameter name (`jbo.doconnectionpooling`) does. The setting of `jbo.doconnectionpooling=false` does not mean that there is no database connection pooling happening. What it means is that the application module is not disconnected from its JDBC connection upon check in back to the application module pool.

If the default setting of `jbo.doconnectionpooling=false` is used, then when an application module instance is created in any pool it acquires a JDBC connection from the appropriate connection pool (based on the JDBC URL in the ADF case, or from the underlying JDBC data source implementation's pool in the case of a JNDI data source name). That application module instance holds onto the JDBC connection object that it acquired from the pool until the application module instance is removed from the application module pool. During its lifetime, that application module instance may service many different users, and ADF worries about issuing rollbacks on the database connection so that different users don't end up getting pending database state confused. By holding onto the JDBC connection, it allows each application module instance to keep its JDBC PreparedStatements's open and usable across subsequent accesses by clients, thereby providing the best performance.

If `jbo.doconnectionpooling=true`, then each time a user session finishes using an application module (typically at the end of each HTTP request), the application module instance disassociates itself with the JDBC connection it was using on that request and it returns it to the JDBC connection pool. The next time that application module instance is used by a user session, it will reacquire a JDBC connection from the JDBC connection pool and use it for the span of time that application module is checked out of the application module pool (again, typically the span of one HTTP request). Since the application module instance "unplugs" itself from the JDBC connection object used to create the PreparedStatements it might have used during the servicing of the current HTTP request, those PreparedStatements are no longer usable on the next HTTP request because they are only valid in the context of the Connection object in which they were created. So, when using the connection pooling mode turned on like this, the trade-off is slightly more JDBC overhead setup each time, in return for using a smaller number of overall database connections.

The key difference is seen when many application module pools are all using the same underlying database user for their application connection.

- If 50 different application module pools each have even just a single application module instance in them, with `jbo.doconnectionpooling=false` there will be 50 JDBC application connections in use. If the application module pooling

- parameters are set such that the application module pools are allowed to shrink to 0 instances after an appropriate instance idle timeout by setting `jbo.ampool.minavailablesize=0`, then when the application module is removed from its pool, it will put back the connection its holding onto.
- In contrast, if 50 different application module pools each have a single application module instance and `jbo.doconnectionpooling=true`, then the amount of JDBC connections in use will depend on how many of those application modules are *simultaneously* being used by different clients. If an application module instance is in the pool and is not currently being used by a user session, then with `jbo.doconnectionpooling=true` it will have released its JDBC connection back to the connection pool and while the application module instance is sitting there waiting for either another user to need it again, or to eventually be cleaned up by the application module pool monitor, it will not be "hanging on" to a JDBC connection.

Performance Tip: Oracle recommends leaving the `jbo.doconnectionpooling` configuration parameter set to `false` for best performance without sacrificing scalability and reliability. Database connection pooling is still achieved through application module pooling. The only exception is when multiple application module pools (and therefore a large number of application module instances) share the same database, making the total available database connections the highest priority.

Highest performance is achieved by not disconnecting the application module instance from its database connection on each check in to the application module pool.

Accordingly, the default setting of the `jbo.doconnectionpooling` configuration parameter is `false`. The pooling of application module instances is already an effective way to optimize resource usage, and the Oracle ADF runtime is more efficient when you do not have to disconnect application module instances from their associated JDBC connection after each release to the pool. Effectively, by pooling the application modules which are related one-to-one with a JDBC connection, you are already achieving a pooling of database connections that is optimal for most Fusion web applications.

In contrast to Oracle's default recommendation, one situation in which it might be appropriate to use database connection pooling is when you have a large number of application module pools all needing to use database connections from the same underlying application user at the database level. In this case, the many application module pools can perhaps economize on the total overall database sessions by sharing a single, underlying database connection pool of JDBC connections, albeit at a loss of efficiency of each one. This choice would be favored only if total overall database sessions is of maximum priority. In this scenario, if a user scrolls through some, but not all rows of a view object's row set, then with `jbo.doconnectionpooling=true`, Oracle ADF will automatically passivate the pending application module state (including current row information) so that the next time the application module is used, the queried view object can be put back into the same current row with the same initial rows fetched. This passivation behavior may reduce performance.

37.8 Database User State and Pooling Considerations

Sometimes you may need to invoke stored procedures to initialize database state related to the current user's session. The correct place to perform this initialization is in an overridden `prepareSession()` method of your application module.

37.8.1 How Often `prepareSession()` Fires When `jbo.doconnectionpooling = false`

The default setting for `jbo.doconnectionpooling` is `false`. This means the application module instance hangs onto its JDBC connection while it's in the application module pool. This is the most efficient setting because the application module can keep its JDBC prepared statements open across application module checkouts/checkins. The application module instance will trigger its `prepareSession()` method each time a new user session begins using it.

37.8.2 How to Set Database User State When `jbo.doconnectionpooling = true`

If you set `jbo.doconnectionpooling` to `true`, then on each checkout of an application module from the pool, that application module pool will acquire a JDBC connection from the database connection pool and use it during the span of the current request. At the end of the request when the application module is released back to the application module pool, that application module pool releases the JDBC connection it was using back to the database connection pool.

It follows that with `jbo.doconnectionpooling` set to `true` the application module instance in the pool may have a completely different JDBC connection each time you check it out of the pool. In this situation, the `prepareSession()` method will fire each time the application module is checked out of the pool to give you a chance to reinitialize the database state.

37.8.3 How to Set Database State

The Fusion web application can set database state on a per-user basis. You typically create a database CONTEXT namespace, associate a PL/SQL procedure with it, and then use the `SYS_CONTEXT()` SQL function to reference values from the context.

[[XREF Example]]

Part VII

Appendices

Part VII contains the following appendices:

- [Appendix A, "Oracle ADF XML Files"](#)
- [Appendix B, "Oracle ADF Binding Properties"](#)
- [Appendix C, "Oracle ADF Permission Grants"](#)
- [Appendix D, "ADF Equivalents of Common Oracle Forms Triggers"](#)
- [Appendix E, "Most Commonly Used ADF Business Components Methods"](#)
- [Appendix F, "ADF Business Components Java EE Design Pattern Catalog"](#)
- [Appendix G, "Performing Common Oracle Forms Tasks in Oracle ADF"](#)

A

Oracle ADF XML Files

This appendix provides reference for the Oracle ADF metadata files that you create in your data model and user interface projects. You may use this information when you want to edit the contents of the metadata these files define.

This appendix includes the following sections:

- [Section A.1, "Introduction to the ADF Metadata Files"](#)
- [Section A.2, "ADF File Overview Diagram"](#)
- [Section A.3, "adfm.xml"](#)
- [Section A.4, "modelProjectName.jpx"](#)
- [Section A.5, "bc4j.xcfg"](#)
- [Section A.6, "DataBindings.cpx"](#)
- [Section A.7, "pageNamePageDef.xml"](#)
- [Section A.8, "adfc-config.xml"](#)
- [Section A.9, "task-flow-definition.xml"](#)
- [Section A.10, "adf-config.xml"](#)
- [Section A.11, "adf-settings.xml"](#)
- [Section A.12, "web.xml"](#)
- [Section A.13, "logging.xml"](#)

A.1 Introduction to the ADF Metadata Files

Metadata files in the Oracle Fusion web application are structured XML files used by the application to:

- Specify the parameters, methods, and return values available to your application's Oracle ADF data control usages
- Create objects in the Oracle ADF binding context and define the runtime behavior of those objects
- Define configuration information about the UI components in JSF and Oracle ADF Faces
- Define application configuration information for the Java EE application server

In the case of ADF bindings, you can use the binding-specific editors to customize the runtime properties of the binding objects. You can open a binding editor when you

display the Structure window for a page definition file and choose **Properties** from the context menu.

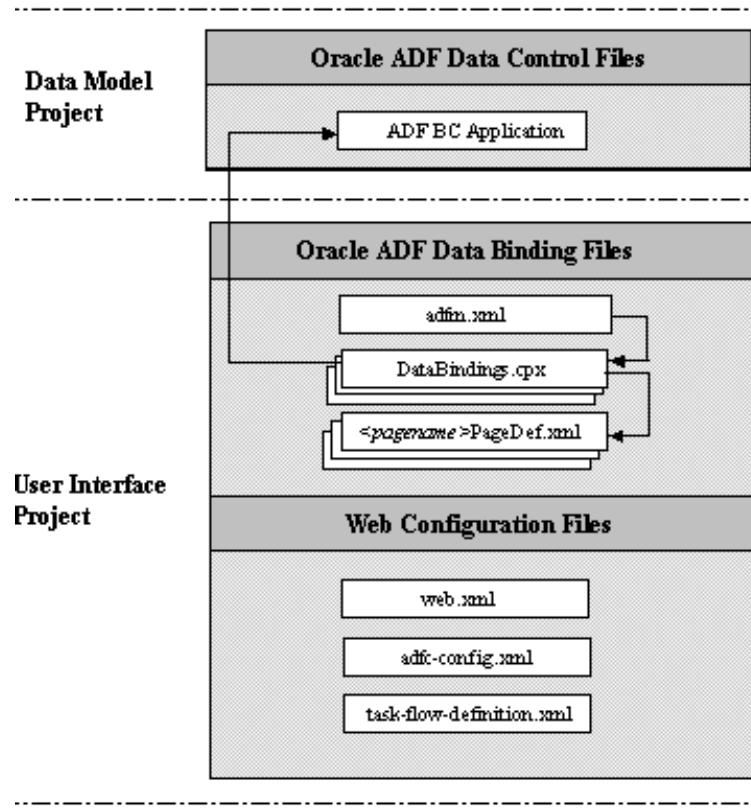
Additionally, you can view and edit the contents of any metadata file in JDeveloper's XML editor. The easiest way to work with these files is through the Structure window and Property Inspector. In the Structure window, you can select an element and in the Property Inspector, you can define attribute values for the element, often by choosing among dropdown menu choices. Use this reference to learn the choices you can select in the case of the Oracle ADF-specific elements.

A.2 ADF File Overview Diagram

The relationship between the Oracle ADF metadata files defines dependencies between the model data and the user interface projects. The dependencies are defined as file references within XML elements of the files.

[Figure A-1](#) illustrates the hierarchical relationship of the XML metadata files that you might work with in a Fusion web application that uses an ADF Business Components application module as a service interface to JSF web pages.

Figure A-1 Oracle ADF File Hierarchy Overview for the Fusion Web Application



A.2.1 Oracle ADF Data Control Files

In an ADF Business Components application, the data control implementation files are contained within the application. The application module and view object XML component descriptor files provide the references for the data control. These files, in conjunction with the bc4j.xcfg file, provide the necessary information for the data control.

A.2.2 Oracle ADF Data Binding Files

These standard XML configuration files for a Fusion web application appear in your user interface project:

- `adfmx.xml`: This file lists the `DataBindings.cpx` file that are available in the current project.

Note: The `web.xml` file no longer contains a `databinding.cpx` entry. This entry is now in the `adfmx.xml` file.

See [Section A.3, "adfmx.xml"](#) for more information.

- `DataBindings.cpx` : This file contains the page map, page definitions references, and data control references. The file is created the first time you create a data binding for a UI component (either from the Structure window or from the Data Controls Panel). The `DataBindings.cpx` file defines the Oracle ADF binding context for the entire application. The binding context provides access to the bindings and data controls across the entire application. The `DataBindings.cpx` file also contains references to the `<pagename>PageDef.xml` files that define the metadata for the Oracle ADF bindings in each web page.

For more information, see [Section A.6, "DataBindings.cpx"](#).

- `<pagename>PageDef.xml`: This is the page definition XML file. This file is created each time you design a new web page using the Data Controls Panel or Structure window. These XML files contain the metadata used to create the bindings that populate the data in the web page's UI components. For every web page that refers to an ADF binding, there must be a corresponding page definition file with binding definitions.

For more information, see [Section A.7, "pageNamePageDef.xml"](#).

A.2.3 Web Configuration Files

These standard XML configuration files required for a JSF application appear in your user interface project:

- `web.xml`: Part of the application's configuration is determined by the contents of its Java EE application deployment descriptor, `web.xml`. The `web.xml` file defines everything about your application that a server needs to know. The file plays a role in configuring the Oracle ADF data binding by setting up the `ADFBindingFilter`. Additional runtime settings include servlet runtime and initialization parameters, custom tag library location, and security settings.

For more information about ADF data binding and JSF configuration options, see [Section A.12, "web.xml"](#).

An ADF Faces application typically uses its own set of configuration files in addition to `web.xml`. For more information, see the "Configuration in `trinidad-config.xml`" section in the *Oracle Fusion Middleware Web User Interface Developer's Guide for Oracle Application Development Framework*.

- `adfc-config.xml` : The configuration file for an ADF unbounded task flow. The configuration file contains metadata about the activities and control flows contained in the unbounded task flow. The default name for this file is `adfc-config.xml`, but an end user can change the name.

For more information, see [Section A.8, "adfc-config.xml"](#).

- `task-flow-definition.xml`: The configuration file for an ADF bounded task flow. The configuration file contains metadata about the activities and control flows contained in the bounded task flow. The default name for this file can be `task-flow-defintion.xml` or whatever an end user specifies in the **Create ADF Task Flow** dialog. The same application can contain multiple task flow definition files.

For more information, see [Section A.9, "task-flow-definition.xml"](#).

A.3 adfm.xml

The `adfm.xml` file contains the classpath-relative paths for the `.cpx`, `.dcx`, `.jpx`, and `.xcfg` files in each design time project that is included in the runtime deployed application. The `adfm.xml` file operates as a dynamically maintained "Registry of Registries" that is used to quickly find all `.cpx`, `.dcx`, `.jpx`, and `.xcfg` files (which are themselves registries of metadata).

The file registry is used extensively by the ADF Library resource catalog browsing implementations, by the ADF model layer design time, and at runtime during merge and discovery.

When a developer creates a binding on a page, JDeveloper adds metadata files (for example, page definitions) in the project source tree. The `adfm.xml` file then notes the location of each.

When a project is built, the `adfm.xml` file is put in `project-root/adfmsrc/META-INF/adfm.xml`. The project-level archive deployment profiles locate the file at `META-INF/adfm.xml`.

At runtime, the application classpath is scanned to build the list of `.cpx` files that comprise the application. The application then loads each `.cpx` as needed to create the binding context. For details about the ADF model layer usage, see [Section 11.3.2, "What Happens When You Use the Data Controls Panel"](#).

Four types of sub registries are recorded by the `adfm.xml` file:

- `DataBindingRegistry (.cpx)`
- `DataControlRegistry (.dcx)`
- `BusinessComponentServiceRegistry (.xcfg)`
- `BusinessComponentProjectRegistry (.jpx)`

A.4 `modelProjectName.jpx`

The `.jpx` file contains configuration information that JDeveloper uses in the design time to allow you to create the Business Components project. It also contains metadata that defines how a shared application module is used at runtime. Because the shared application module can be accessed by any Business Components project in the same application, JDeveloper maintains the scope of the shared application module in the Business Components project configuration file.

This file is saved in the `src` directory of the project. For example, if you look at the `StoreFrontService.jpx` file in the `./src/model` subdirectory of the Fusion Order Demo application's `StoreFrontService` project, you will see the `SharedLookupService` application module's usage definition. For details about the shared application module usage, see [Section 10.2.2, "What Happens When You Define a Shared Application Module"](#).

[Example A-2](#) displays a sample default `.jpx` file.

Example A-1 Sample .jpx File

```

<JboProject
    xmlns="http://xmlns.oracle.com/bc4j"
    Name="StoreFrontService"
    Version="11.1.1.49.73"
    SeparateXMLFiles="true"
    PackageName=" "
    <DesignTime>
        <Attr Name="_appModuleNames0"
            Value="oracle.fodemo.storefront.lookups.LookupServiceAM" />
        <Attr Name="_domainNames0"
            Value="oracle.fodemo.storefront.entities.formatters.UppercaseOracleStyleDate" />
        <Attr Name="_jprName" Value="..../StoreFrontService.jpr" />
        <Attr Name="_appModuleNames1"
            Value="oracle.fodemo.storefront.store.service.StoreServiceAM" />
        <Attr Name="_NamedConnection" Value="FOD" />
    </DesignTime>
    <Containee
        Name="links"
        FullName="oracle.fodemo.storefront.account.queries.links.links"
        ObjectType="JboPackage">
        <DesignTime>
            <Attr Name="_VO" Value="true" />
            <Attr Name="_VL" Value="true" />
        </DesignTime>
    </Containee>
    <Containee
        Name="queries"
        FullName="oracle.fodemo.storefront.account.queries.queries"
        ObjectType="JboPackage">
        <DesignTime>
            <Attr Name="_VO" Value="true" />
        </DesignTime>
    </Containee>
    <Containee
        Name="associations"
        FullName="oracle.fodemo.storefront.entities.associations.associations"
        ObjectType="JboPackage">
        <DesignTime>
            <Attr Name="_AS" Value="true" />
        </DesignTime>
    </Containee>
    <Containee
        Name="entities"
        FullName="oracle.fodemo.storefront.entities.entities"
        ObjectType="JboPackage">
        <DesignTime>
            <Attr Name="_EO" Value="true" />
        </DesignTime>
    </Containee>
    <Containee
        Name="formatters"
        FullName="oracle.fodemo.storefront.entities.formatters.formatters"
        ObjectType="JboPackage">
        <DesignTime>
            <Attr Name="_DO" Value="true" />
        </DesignTime>
    </Containee>
    <Containee
        Name="lookups"

```

```

FullName="oracle.fodemo.storefront.lookups.lookups"
ObjectType="JboPackage">
<DesignTime>
  <Attr Name="_VO" Value="true"/>
  <Attr Name="_AM" Value="true"/>
</DesignTime>
</Containee>
<Containee
  Name="links"
  FullName="oracle.fodemo.storefront.store.queries.links.links"
  ObjectType="JboPackage">
<DesignTime>
  <Attr Name="_VL" Value="true"/>
</DesignTime>
</Containee>
<Containee
  Name="queries"
  FullName="oracle.fodemo.storefront.store.queries.queries"
  ObjectType="JboPackage">
<DesignTime>
  <Attr Name="_VO" Value="true"/>
</DesignTime>
</Containee>
<Containee
  Name="service"
  FullName="oracle.fodemo.storefront.store.service.service"
  ObjectType="JboPackage">
<DesignTime>
  <Attr Name="_AM" Value="true"/>
</DesignTime>
</Containee>
<AppModuleUsage
  Name="SharedLookupService"
  FullName="oracle.fodemo.storefront.lookups.LookupServiceAM"
  ConfigurationName="oracle.fodemo.storefront.lookups.null"
  SharedScope="1"/>
</JboProject>

```

A.5 bc4j.xcfg

The `bc4j.xcfg` file contains metadata information about application module names, the database connection used by the application module, and the runtime parameters the user has configured for the application module.

The `bc4j.xcfg` file is located in the `./common` subdirectory relative to the application module's XML component definition. All of the configurations for all of the application modules in a single Java package are saved in that same file. For example, if you look at the `bc4j.xcfg` file in the `./classes/oracle/fodemo/storefront/store/service/common` directory of the Fusion Order Demo application's StoreFront project, you will see the three named configurations for its `StoreServiceAM` application module. For details about editing the configurations, see [Section 9.3.3, "How to Change Your Application Module's Runtime Configuration"](#) and [Section 37.3, "Setting Pool Configuration Parameters"](#).

[Example A-2](#) displays a sample `bc4j.xcfg` file from the Fusion Order Demo application.

Example A-2 Sample bc4j.xcfg File

```
<BC4JConfig version="11.0" xmlns="http://xmlns.oracle.com/bc4j/configuration">
  < AppModuleConfigBag>
    < AppModuleConfig name="StoreServiceAMLocal"
      ApplicationName="oracle.fodemo.storefront.store.service.StoreServiceAM"
      DeployPlatform="LOCAL"
      JDBCName="FOD"
      jbo.project="StoreFrontService"
      java.naming.factory.initial="oracle.jbo.common.JboInitialContextFactory">
      <Database jbo.locking.mode="optimistic"/>
      <Security AppModuleJndiName="oracle.fodemo.storefront.store.service.StoreServiceAM" />
    </ AppModuleConfig>
    < AppModuleConfig name="StoreServiceAMShared"
      ApplicationName="oracle.fodemo.storefront.store.service.StoreServiceAM"
      DeployPlatform="LOCAL"
      JDBCName="FOD"
      jbo.project="StoreFrontService"
      java.naming.factory.initial="oracle.jbo.common.JboInitialContextFactory">
      <AM-Pooling jbo.ampool.isuseexclusive="false" jbo.ampool.maxpoolsize="1" />
      <Security AppModuleJndiName="oracle.fodemo.storefront.store.service.StoreServiceAM" />
    </ AppModuleConfig>
    < AppModuleConfig name="StoreFrontService"
      ApplicationName="oracle.fodemo.storefront.store.service.StoreServiceAM"
      DeployPlatform="LOCAL"
      JDBCName="FOD"
      jbo.project="StoreFrontService"
      java.naming.factory.initial="oracle.jbo.common.JboInitialContextFactory">
      <AM-Pooling jbo.ampool.resetnontransactionalstate="false" />
      <Database jbo.locking.mode="optimistic" />
      <Security AppModuleJndiName="oracle.fodemo.storefront.store.service.StoreServiceAM" />
      <Custom JDBCDataSource="jdbc/FODDS" />
    </ AppModuleConfig>
  </ AppModuleConfigBag>
</BC4JConfig>
```

A.6 DataBindings.cpx

The DataBindings.cpx file is created in the user interface project the first time you drop a data control usage onto a web page in the HTML visual editor. The DataBindings.cpx file defines the Oracle ADF binding context for the entire application and provides the metadata from which the Oracle ADF binding objects are created at runtime. It is used extensively by the ADF Library Resource Palette browsing implementations, and also by the .cpx and .dcx design time and runtime merge and discovery. When you insert a databound UI component into your document, the page will contain binding expressions that access the Oracle ADF binding objects at runtime.

If you are familiar with building Fusion web applications in earlier releases of JDeveloper, you'll notice that the .cpx file no longer contains all the information copied from the DataControls.dcx file, but only a reference to it. If you need to make changes to the .cpx file, you must edit the DataControls.dcx file.

The DataBindings.cpx file appears in the /src directory of the user interface project. When you double-click the file node, the binding context description appears in the XML source editor. (To edit the binding context parameters, use the Property Inspector and select the desired parameter in the Structure window.)

A.6.1 DataBindings.cpx Syntax

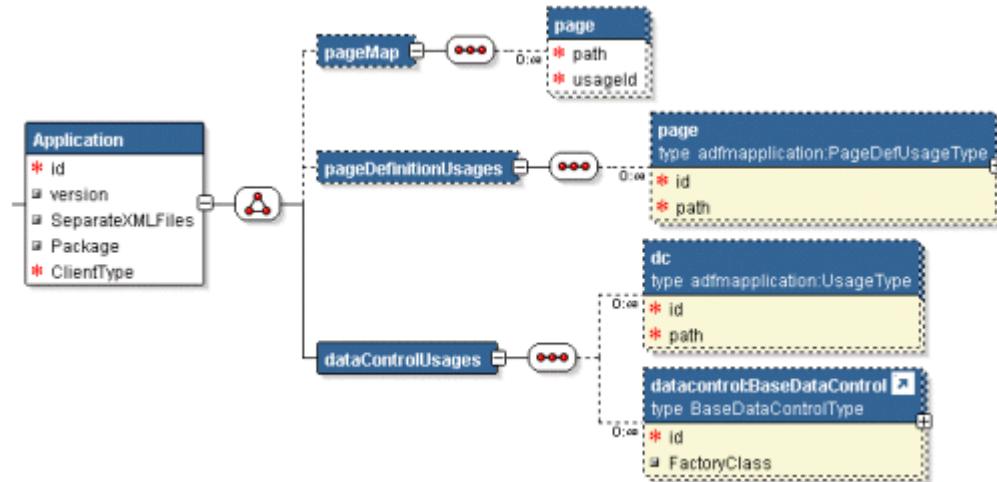
The top level element of the DataBindings . cpx file is <DataControlConfigs>:

```
<?xml version = '1.0' encoding = 'UTF-8'?>
<BC4JConfig version="11.0" xmlns="http://xmlns.oracle.com/bc4j/configuration">
```

where the XML namespace attribute (`xmlns`) specifies the URI to which the data controls bind at runtime. Only the package name is editable; all other attributes should have the values shown.

[Figure A-2](#) displays the child element hierarchy of the <DataControlConfigs> element. Note that each business service for which you have created a data control will have its own <dataControlUsages> definition.

Figure A-2 Schema for the Structure Definition of the DataBindings.cpx File



The child elements have the following usages:

- <definitionFactories> registers a factory class to create the ADF binding objects associated with a particular namespace at runtime. The factory class is specific to the namespace associated with the type of ADF binding (for instance, a task flow binding).
- <pageMap> maps all user interface URLs and the corresponding page definition usage name. This map is used at runtime to map a URL to its page definition.
- <pageDefinitionUsages> maps a page definition usage (BindingContainer instance) name to the corresponding page definition. The `id` attribute represents the usage ID. The `path` attribute represents the full path to the page definition.
- <dataControlUsages> declares a list of data control usages (shortnames) and corresponding path to the data control definition entries in the `.dcx` or `.xcfg` file.

[Table A-1](#) describes the attributes of the DataBindings . cpx elements.

Table A-1 Attributes of the DataBindings.cpx File Elements

Element Syntax	Attributes	Attribute Description
<definitionFactories> <factory/> </definitionFactories>	nameSpace	A URI. Identifies the location of the executable elements in the page definition usage.
	className	The fully qualified class name. Identifies the location of the factory class that creates the page definition usage objects.
<pageMap> <page /> </pageMap>	path	The full directory path. Identifies the location of the user interface page.
	usageId	A unique qualifier. Names the page definition ID that appears in the ADF page definition file. The ADF binding servlet looks at the incoming URL requests and checks that the bindings variable is pointing to the ADF page definition associated with the URL of the incoming HTTP request.
<pageDefinitionUsages> <page /> </pageDefinitionUsages>	id	A unique qualifier. References the page definition ID that appears in the ADF page definition file.
	path	The fully qualified package name. Identifies the location of the user interface page's ADF page definition file.
<dataControlUsages> <dc... /> </dataControlUsages>	id	A unique qualifier. Identifies the data control usage as is defined in the DataControls.dcx file.
	path	The fully qualified package name. Identifies the location of the data control.

A.6.2 DataBindings.cpx Sample

[Example A-3](#) shows the syntax for the DataBindings.cpx file in the Fusion Order Demo application.

The ADF executable definition factory (factory element) is named by a className attribute and is associated with a namespace. At runtime, the factory class creates the executable definition objects that leads to the creation of the binding objects for the ADF binding container associated with a particular page definition. The factory locates the page definition through two DataBindings.cpx file elements: the pageMap element that maps the page URL to the page definition ID (usageId attribute) assigned at design time and the pageDefinitionUsages element that maps the ID to the location of the page definition from the project or project classpath.

Additionally, the ADF Business Components data control (BC4JDataControl element) is named by the id attribute. The combination of the Package attribute and the Configuration attribute is used to locate the bc4j.xcfg file in the ./common subdirectory of the indicated package. The configuration contains the information of the application module name and all the runtime parameters the user has configured.

Example A-3 Sample DataBindings.cpx File

```
<Application xmlns="http://xmlns.oracle.com/adfm/application"
             version="11.1.1.44.61" id="DataBindings" SeparateXMLFiles="false"
             Package="oracle.fodemo.storefront" ClientType="Generic"
             ErrorHandlerClass="oracle.fodemo.frmwkext.FODCustomErrorHandler">
  <definitionFactories>
```

```

<factory nameSpace="http://xmlns.oracle.com/adf/controller/binding"
        className="oracle.adf.controller.internal.binding.
TaskFlowBindingDefFactoryImpl" />
<factory nameSpace="http://xmlns.oracle.com/adfm/dvt"
        className="oracle.adfinternal.view.faces.dvt.model.binding.
FacesBindingFactory" />
</definitionFactories>
<pageMap>
    <page path="/home.jspx" usageId="homePageDef">
    ...
</pageMap>
<pageDefinitionUsages>
    <page id="homePageDef"
        path="oracle.fodemo.storefront.pageDefs.homePageDef" />
    ...
</pageDefinitionUsages>
<dataControlUsages>
    <BC4JDataControl id="StoreServiceAMDataControl"
        Package="oracle.fodemo.storefront.store.service"
        FactoryClass="oracle.adf.model.bc4j.DataControlFactoryImpl"
        SupportsTransactions="true" SupportsFindMode="true"
        SupportsRangesize="true" SupportsResetState="true"
        SupportsSortCollection="true"
        Configuration="StoreServiceAMLocalWeb" syncMode="Immediate"
        xmlns="http://xmlns.oracle.com/adfm/datacontrol" />
    ...
</dataControlUsages>
</Application>

```

A.7 pageNamePageDef.xml

The `pageNamePageDef.xml` files are created each time you insert a databound component into a web page using the Data Controls Palette or Structure window. These XML files define the Oracle ADF binding container for each web page in the application. The binding container provides access to the bindings within the page. You will have one XML file for each databound web page.

Caution: The `DataBindings.cpx` file maps JSF pages to their corresponding page definition files. If you change the name of a page definition file or a JSF page, JDeveloper does *not* automatically refactor the `DataBindings.cpx` file. You must manually update the page mapping in the `DataBindings.cpx` file.

The `PageDef.xml` file appears in the `/src/view` directory of the ViewController project. The Application Navigator displays the file in the view package of the Application Sources node. When you double-click the file node, the page description appears in the XML source editor. To edit the page description parameters, use the Property Inspector and select the desired parameter in the Structure window.

For more information, see [Chapter 11.6, "Working with Page Definition Files"](#).

There are important differences in how the page definitions are generated for methods that return a single value and a collection.

A.7.1 PageDef.xml Syntax

The top-level element of the `PageDef.xml` file is `<pageDefinition>`:

```
<pageDefinition xmlns="http://xmlns.oracle.com/adfm/uimodel"
    version="10.1.3.35.83" id=<pagename>PageDef"
    Package="oracle.fod.view.pageDefs">
```

where the XML namespace attribute (`xmlns`) specifies the URI to which the ADF binding container binds at runtime. Only the package name is editable; all other attributes should have the values shown.

Example A-4 displays the child element hierarchy of the `<pageDefinition>` element. Note that each business service for which you have created a data control will have its own `<AdapterDataControl>` definition.

Example A-4 PageDef.xml Element Hierarchy

```
<?xml version="1.0" encoding="UTF-8" ?>
<pageDefinition>
    <parameters>
        ...
    </parameters>
    <executables>
        ...
    </executables>
    <bindings>
        ...
    </bindings>
</pageDefinition>
```

The child elements have the following usages:

- `<parameters>` defines page-level parameters that are EL accessible. These parameters store information local to the web page request and may be accessed in the binding expressions.
- `<executables>` defines the list of items (methods, view objects, and accessors) to execute during the `prepareModel` phase of the ADF page lifecycle. Methods to be executed are defined by `<methodIterator>`. The lifecycle performs the execute in the sequence listed in the `<executables>` section. Whether or not the method or operation is executed depends on its `refresh` or `refreshCondition` attribute value. Built-in operations on the data control are defined by:
 - `<page>` - definition for a nested page definition (binding container)
 - `<iterator>` - definition to a named collection in DataControls
 - `<accessorIterator>` - definition to get an accessor in a data control hierarchy
 - `<methodIterator>` - definition to get to an iterator returned by an invoked method defined by a `methodAction` in the same file
 - `<variableIterator>` - internal iterator that contains variables declared for the binding container
 - `<invokeAction>` - definition of which method to invoke as an executable
- `<bindings>` refers to an entry in `<executables>` to get to the collection from which bindings extract/submit attribute level data.

Table A-2 describes the attributes of the top-level `<pageDefinition>` element.

Table A-2 Attributes of the PageDef.xml File <pageDefinition> Element

Element Syntax	Required	Attributes	Attribute Description
<pageDefinition>		ControllerClass	Fully qualified class name to create when controller requests a PageController object for this bindingContainer.
		EnableTokenValidation	Enables currency validation for this bindingContainer when a postback occurs. This is to confirm that the web tier state matches the state that particular page was rendered with.
		FindMode	FindMode is for legacy (10.1.2) use only and indicates whether this bindingContainer should start out in findMode when initially prepared.
		MsgBundleClass	Fully qualified package name. Identifies the class which contains translation strings for any bindings.
		Viewable	An EL expression that should resolve at runtime to whether this binding and the associated component should be rendered or not.

Table A-3 describes the attributes of the child element of <parameters>.

Table A-3 Attributes of the PageDef.xml File <parameters> Element

Element Syntax	Required	Attributes	Attribute Description
<parameter>		evaluate	Specifies when the parameter should be evaluated: eachUse, firstUse, or inPrepareModel.
		id	Unique identifier. May be referenced by ADF bindings.
		option	Indicates the usage of the variable within the binding container: <ul style="list-style-type: none"> ▪ Final indicates that this parameter cannot be passed in by a usage of this binding container. It must use the default value in the definition. ▪ Optional indicates that the variable value need not be provided. ▪ Mandatory indicates that the variable value must be provided or a binding container exception will be thrown.
		readonly	Indicates whether the parameter value may be modified or not. Set to true when you do not want the application to modify the parameter value.
		value	A default value, which can be an EL expression.

Table A-4 describes the attributes of the PageDef.xml <executables> elements.

Table A-4 Attributes of the PageDef.xml File <executables> Element

Element Syntax	Required	Attributes	Attribute Description
<accessorIterator>		Accessor Binds	Specifies any other accessor defined by this binding. Specifies the view or action to which the iterator is bound.
		BeanClass	Identifies the Java type of beans in the associated iterator or collection.
		CacheResults	If true, manages the data collection between requests.
		ChangeEventRate	Specifies the rate of events when a component is wired to data via this iterator and is in polling event mode.
		DataControl	Interprets and returns the collection referred to by this iterator binding.
		id	Unique identifier. May be referenced by any ADF value binding.
		MasterBinding	Reference to the methodIterator (or iterator) that binds the data collection that serves as the master to the accessor iterator's detail collection.
		ObjectType	Used for ADF Business Components only. A boolean value determines whether the collection is an object type or not.
		RangeSize	Specifies the number of data objects in a range to fetch from the bound collection. The range defines a window you can use to access a subset of the data objects in the collection. By default, the range size is set to a range that fetches just ten data objects. Use RangeSize when you want to work with an entire set or when you want to limit the number of data objects to display in the page. Note that the values -1 and 0 have specific meaning: the value -1 returns all available objects from the collection, while the value 0 will return the same number of objects as the collection uses to retrieve from its data source.

Table A–4 (Cont.) Attributes of the PageDef.xml File <executables> Element

Element Syntax	Required	Attributes	Attribute Description
		Refresh	<p>Determines when and whether the executable should be invoked. Set one of the following properties as required:</p> <ul style="list-style-type: none"> ■ <code>always</code> - causes the executable to be invoked each time the binding container is prepared. This will occur when the page is displayed and when the user submits changes, or when the application posts back to the page. ■ <code>deferred</code> (default for ADF faces applications) - refresh occurs when another binding requires or refers to this executable. Since refreshing an executable may be a performance concern, you can set the refresh to occur only if <code>deferred</code> is used in a binding that is being rendered. ■ <code>ifNeeded</code> (default for all view technologies other than ADF Faces) - whenever the framework needs to refresh the executable because it has not been refreshed to this point. For example, when you have an accessor hierarchy such that a detail is listed first in the page definition, the master could be refreshed twice (once for the detail and again for the master's iterator). Using <code>ifNeeded</code> gives the mean to avoid duplicate refreshes. This is the default behavior for executables. ■ <code>never</code> - when the application itself will call <code>refresh</code> on the executable during one of the controller phases and does not want the framework to refresh it at all. ■ <code>prepareModel</code> - causes the executable to be invoked each time the page's binding container is prepared. ■ <code>prepareModelIfNeeded</code> - causes the executable to be invoked during the <code>prepareModel</code> phase if this executable has not been refreshed to this point. See also <code>ifNeeded</code> above. ■ <code>renderModel</code> - causes the executable to be invoked each time the page is rendered. ■ <code>renderModelIfNeeded</code> - causes the executable to be invoked during the page's <code>renderModel</code> phase on the condition that it is needed. See also <code>ifNeeded</code> above.

Table A-4 (Cont.) Attributes of the PageDef.xml File <executables> Element

Element Syntax	Required	Attributes	Attribute Description
		RefreshCondition	An EL expression that when resolved, determines when and whether the executable should be invoked. For example, \${!bindings.findAllServiceRequestIter.findModel} resolves the value of the findMode on the iterator in the ADF binding context AllServiceRequest. Hint: Use the Property Inspector to create expressions from the available objects of the binding context (bindings namespace) or binding context (data namespace), JSF managed beans, and JSP objects.
		RefreshAfter	Specifies the condition after which the page should be refreshed.
		Sortable	Specifies whether the iterator is sortable or not.

Table A–4 (Cont.) Attributes of the PageDef.xml File <executables> Element

Element Syntax	Required	Attributes	Attribute Description
<invokeAction>		Binds	<p>Determines the action to invoke. This may be on any actionBinding. Additionally, in the case, of the EJB session facade data control, you may bind to the finder method exposed by the data control. Built-in actions supported by the EJB session facade data control include:</p> <ul style="list-style-type: none"> ▪ Execute executes the bound action defined by the data collection. ▪ Find retrieves a data object from a collection. ▪ First navigates to the first data object in the data collection range. ▪ Last navigates to the first data object in the data collection range. ▪ Next navigates to the first data object in the data collection range. If the current range position is already on the last data object, then no action is performed. ▪ Previous navigates to the first data object in the data collection range. If the current position is already on the first data object, then no action is performed. ▪ setCurrentRowWithKey passes the row key as a String converted from the value specified by the input field. The row key is used to set the currency of the data object in the bound data collection. When passing the key, the URL for the form will not display the row key value. You may use this operation when the data collection defines a multipart attribute key. ▪ setCurrentRowWithValue is used as above, but when you want to use a primary key value instead of the stringified key.
		id	Unique identifier. May be referenced by any ADF action binding.
		Refresh	See Refresh for <accessorIterator>.
		RefreshCondition	See RefreshCondition for <accessorIterator>.
<iterator> and <methodIterator>		BeanClass	Identifies the Java type of beans in the associated iterator or collection.
		BindingClass	This is for backward compatibility to indicate which class implements the runtime for this binding definition. Not used in current JDeveloper release.
		Binds	See Binds for <invokeAction>.

Table A-4 (Cont.) Attributes of the PageDef.xml File <executables> Element

Element Syntax	Required	Attributes	Attribute Description
		CacheResults	See CacheResults for <accessorIterator>.
		ChangeEventRate	Specifies the rate of events when a component is wired to data via this iterator and is in polling event mode.
		DataControl	Name of the DataControl usage in the bindingContext (.cpk) which this iterator is associated with.
		DefClass	Used internally by ADF.
		id	Unique identifier. May be referenced by any ADF value binding.
		ObjectType	Not used by EJB session facade data control (used by ADF Business Components only).
		RangeSize	See RangeSize for <accessorIterator>.
		Refresh	See Refresh for <accessorIterator>.
		RefreshAfter	Specifies the condition after which the page should be refreshed.
		RefreshCondition	See RefreshCondition for <accessorIterator>.
<page> and <variableIterator>		id	Unique identifier. In the case of <page>, refers to nested page or region that is included in this page. In the case of the <variableIterator> executable, the identifier may be referenced by any ADF value binding.
		ChangeEventRate	Specifies the rate of events when a component is wired to data via this iterator and is in polling event mode.
		path	Used by <page> executable only. Advanced, a fully qualified path that may reference another page's binding container.
		Refresh	See Refresh for <accessorIterator>.
		RefreshAfter	Specifies the condition after which the page should be refreshed.
		RefreshCondition	See RefreshCondition for <accessorIterator>.

Table A-5 describes the attributes of the PageDef.xml <bindings> element.

Table A–5 Attributes of the PageDef.xml File <bindings> Element

Element Syntax	Required	Attributes	Attribute Description
<action>		Action	Fully qualified package name. Identifies the class for which the data control is created. In the case of the EJB session facade, this is the session bean.
		BindingClass	This is for backward compatibility to indicate which class implements the runtime for this binding definition. This is used by earlier versions of JDeveloper.
		DataControl	Name of the DataControl usage in the bindingContext (.cpk) which this iteratorBinding or actionBinding is associated with.
		Execute	Used by default when you drop an operation from the Data Controls Panel in the automatically configured ActionListener property. It results in executing the action binding's operation at runtime.
		InstanceName	Specifies the instance name for the action.
		IterBinding	Specifies the iteratorBinding instance in this bindingContainer to which this binding is associated.
		Outcome	Use if you want to use the result of a method action binding (once converted to a String) as a JSF navigation outcome name.
<attributeValues>		ApplyValidation	Set to true by default. When true, controlBinding executes validators defined on the binding. You can set to false in the case of ADF Business Components, when running in local mode and the same validators are already defined on the corresponding attribute.
		BindingClass	This is for backward compatibility to indicate which class implements the runtime for this binding definition. This is used by earlier versions of JDeveloper.
		ControlClass	Used internally by ADF.
		CustomInputHandler	This is the class name for a oracle.jbo.uicli.binding.JUCtrlValueHandler implementation that is used to process the inputValue for a given value binding.
		DefClass	Used internally by ADF.
		id	Unique identifier. May be referenced by any ADF action binding.
		IterBinding	Refers to the iteratorBinding instance in this bindingContainer to which this binding is associated.
		NullValueId	Refers to the entry in the message bundle for this bindingContainer that contains the String to indicate the null value in a list display.

Table A-5 (Cont.) Attributes of the PageDef.xml File <bindings> Element

Element Syntax	Required	Attributes	Attribute Description
<button>		ApplyValidation	Set to true by default. When true, controlBinding executes validators defined on the binding. You can set to false in the case of ADF Business Components, when running in local mode and when the same validators are already defined on the corresponding attribute.
		BindingClass	This is for backward compatibility to indicate which class implements the runtime for this binding definition. This is used by earlier versions of JDeveloper.
		BoolVal	Identifies whether the value at the zero index in the static value list in this boolean list binding represents true or false.
		ControlClass	Used internally by ADF.
		CustomInputHandler	This is the class name for a oracle.jbo.uicli.binding.JUCtrlValueHandler implementation that is used to process the inputValue for a given value binding.
		DefClass	Used internally by ADF.
		id	Unique identifier. May be referenced by any ADF action binding.
		IterBinding	Refers to the iteratorBinding instance in this bindingContainer to which this binding is associated.
		ListIter	Refers to the iteratorBinding that is associated with the source list of this listBinding.
		ListOperMode	Determines whether this list binding is for navigation, contains a static list of values or is an LOV type list.
		NullValueFlag	Describes whether this list binding has a null value and, if so, whether it should be displayed at the beginning or the end of the list.
		NullValueId	Refers to the entry in the message bundle for this bindingContainer that contains the String to indicate the null value in a list display.
<ganttDataMap>			Maps the data binding XML for an ADF Faces gantt component.
<gaugeDataMap>			Maps the data binding XML for an ADF Faces gauge component.
<graph>		ApplyValidation	Set to true by default. When true, controlBinding executes validators defined on the binding. You can set to false in the case of ADF Business Components, when running in local mode and when the same validators are already defined on the corresponding attribute.

Table A-5 (Cont.) Attributes of the PageDef.xml File <bindings> Element

Element Syntax	Required	Attributes	Attribute Description
		BindingClass	This is for backward compatibility to indicate which class implements the runtime for this binding definition. This is used by earlier versions of JDeveloper.
		BoolVal	Identifies whether the value at the zero index in the static value list in this boolean list binding represents true or false.
		ChildAccessorName	The name of the accessor to invoke to get the next level of nodes for a given hierarchical node type in a tree.
		ControlClass	Used internally by ADF.
		CustomInputHandler	This is the class name for a oracle.jbo.uicli.binding.JUCtrlValueHandler implementation that is used to process the inputValue for a given value binding.
		DefClass	Used internally by ADF.
		GraphPropertiesFile Name	An XML file that specifies the type of graph to use, for example, pie chart or bar graph. This XML file can be used to customize the visual properties of the graph. It contains graph attributes such as title, subtitle, footnote, graph type, legend area, and plot area. The default filename is BIGraphDef.xml.
		GroupLabel	For master-detail forms, specifies the attribute that will be used to group data.
		id	Unique identifier. May be referenced by any ADF action binding.
		IterBinding	Refers to the iteratorBinding instance in this bindingContainer to which this binding is associated.
		NullValueId	Refers to the entry in the message bundle for this bindingContainer that contains the String to indicate the null value in a list display.
		SeriesLabel	Defines the attribute, based on which data will be clubbed.
		SeriesType	Determines whether graph is for Single View(SINGLE_SERIES), or for MASTER_DETAIL.
<graphDataMap>			Wraps the data binding XML for an ADF Faces graph component.
<list>		ApplyValidation	Set to true by default. When true, controlBinding executes validators defined on the binding. You can set to false in the case of ADF Business Components, when running in local mode and when the same validators are already defined on the corresponding attribute.

Table A-5 (Cont.) Attributes of the PageDef.xml File <bindings> Element

Element Syntax	Required	Attributes	Attribute Description
		BindingClass	This is for backward compatibility to indicate which class implements the runtime for this binding definition. This is used by earlier versions of JDeveloper.
		ControlClass	Used internally by ADF.
		CustomInputHandler	This is the class name for a oracle.jbo.uicli.binding.JUCtrlValueHandler implementation that is used to process the inputValue for a given value binding.
		DefClass	Used internally by ADF.
		id	Unique identifier. May be referenced by any ADF action binding.
		IterBinding	Refers to the iteratorBinding instance in this bindingContainer to which this binding is associated.
		ListIter	Refers to the iteratorBinding that is associated with the source list of this listBinding.
		ListOperMode	Determines whether this list binding is for navigation, contains a static list of values, or is an LOV type list.
		MRUCount	Specifies the number of items to display in a choice list when you want to provide a shortcut for the end-user to display their most recent selections. For example, a form might display a choice list of supplier ID values to drive a purchase order form. In this case, you can allow users to select from a list of their most recently view suppliers, where the number of supplier choices is determined by the count you enter. The default for the choice list is to display all values for the attribute and is specified by the count 0 (zero)."
		MRUID	Specifies the String that will be the discriminator line for the MRU list.
		NullValueFlag	Describes whether this list binding has a null value and, if so, whether it should be displayed at the beginning of the list or the end.
		NullValueId	Refers to the entry in the message bundle for this bindingContainer that contains the String to indicate the null value in a list display.
		StaticList	Defines a static list of values that will be rendered in the bound list component.
<mapThemeDataMap>			Wraps the data binding XML for an ADF Faces pivot table component.
<methodAction>		Action	Fully qualified package name. Identifies the class for which the data control is created. In the case of the EJB session facade, this is the session bean.

Table A–5 (Cont.) Attributes of the PageDef.xml File <bindings> Element

Element Syntax	Required	Attributes	Attribute Description
		BindingClass	This is for backward compatibility to indicate which class implements the runtime for this binding definition. This is used by earlier versions of JDeveloper.
		ClassName	This is the class to which the method being invoked belongs.
		DataControl	Name of the DataControl usage in the bindingContext (.cpx) which this iteratorBinding or actionBinding is associated with.
		DefClass	Used internally by ADF.
		id	Unique identifier. May be referenced by any ADF action binding.
		InstanceName	A dot-separated EL path to a Java object instance on which the associated method is to be invoked.
		IsLocalObjectReference	Set to true if the instanceName contains an EL path relative to this bindingContainer.
		IsViewObjectMethod	Set to true if the instanceName contains an instance path relative to the associated data control's application module.
		MethodName	Indicates the name of the operation on the given instance or class that needs to be invoked for this methodActionBinding.
		RequiresUpdateModel	Whether this action requires that the model be updated before the action is to be invoked.
		ReturnName	The EL path of the result returned by the associated method.
<pivotTableDataMap>			Wraps the data binding XML for an ADF Faces pivot table component.
<table> and <tree>		ApplyValidation	Set to true by default. When true, controlBinding executes validators defined on the binding. You can set to false in the case of ADF Business Components, when running in local mode and when the same validators are already defined on the corresponding attribute.
		BindingClass	This is for backward compatibility to indicate which class implements the runtime for this binding definition. This is used by earlier versions of JDeveloper.
		CollectionModel	Accesses the CollectionModel object, the data model that is used by ADF table components. A table's value is bound to the CollectionModel attribute. The table wraps the result set from the iterator binding in a CollectionModel object. The CollectionModel attribute allows each item in the collection to be available within the table component using the var attribute.
		ControlClass	Used internally for testing purposes.

Table A-5 (Cont.) Attributes of the PageDef.xml File <bindings> Element

Element Syntax	Required	Attributes	Attribute Description
		CustomInputHandler	This is the class name for a oracle.jbo.uicli.binding.JUCtrlValueHandler implementation that is used to process the inputValue for a given value binding.
		DefClass	Used internally by ADF.
		DiscrValue	Indicates the discriminator value for a hierarchical type binding (type definition for a tree node). This value is used to determine whether a given row in a collection being rendered in a polymorphic tree binding should be rendered using the containing hierarchical type binding.
		id	Unique identifier. May be referenced by any ADF action binding.
		IterBinding	Refers to the iteratorBinding instance in this bindingContainer to which this binding is associated.
		TreeModel	The data model used by ADF Tree components. TreeModel extends CollectionModel to add support for container rows. Rows in the TreeModel may (recursively) contain other rows.

A.8 adfc-config.xml

The default name for an ADF unbounded task flow's XML source file is `adfc-config.xml`. Each Fusion web application optionally contains a single ADF unbounded task flow. The `adfc_config.xml` file contains activities, control flow rules, and managed beans interacting to allow a user to complete a task.

The `adfc-config.xml` file is located in the `/public_html/WEB-INF` directory relative to the ADF application's ViewController project.

For more information, see [Section 13.1.2.1, "ADF Unbounded Task Flows"](#).

A.9 task-flow-definition.xml

The XML source file for an ADF bounded task flow is called a *task flow definition*. The default name for this source file is taken from the value specified in the **Task Flow ID** field of the Create ADF Task Flow wizard. The name of this file can be `task-flow-definition.xml`. A Fusion web application can contain one to many ADF bounded task flows, each with its own task flow definition. For more information, see [Section 13.1.2.1, "ADF Unbounded Task Flows"](#).

Note that you can open the XSD file in JDeveloper to see this file in the schema viewer. In JDeveloper, open `task-flow-definition_1_0.xsd` from the zip file at: Oracle Fusion Middleware Web User Interface Developer's Guide for Oracle Application Development Framework <install>/adfc/src/adf-controller-api.zip.

A.10 adf-config.xml

The `adf-config.xml` file specifies application-level settings that are usually determined at deployment and are often changed at runtime. You can use a

deployment profile to specify settings that are used at application deploy time. Use an mbean registered for the component configuration to specify settings used at runtime.

This file is located in the directory <root>/ .adf/meta-inf/. At runtime, the file is loaded from /meta-inf/adf-config.xml. If more than one /meta-inf/adf-config.xml file is found when loading, all loading of the files stops and a warning is logged.

A.11 adf-settings.xml

The adf-settings.xml file holds project-level and library-level settings such as ADF Faces help providers, ADF Controller phase listeners, and webcache project-level configuration information.

The configuration settings for adf-settings.xml are fixed and cannot be changed during or after application deployment. There can be multiple adf-settings.xml files in an application. ADF settings file users are responsible for merging the contents of their configuration.

Example A-5 Sample adf-settings.xml File

```
<adf-settings xmlns="http://xmlns.oracle.com/adf/settings"
    xmlns:wap="http://xmlns.oracle.com/adf/share/http/config" >
    <wap:adf-web-config xmlns="http://xmlns.oracle.com/adf/share/http/config">
        <web-app-root    rootName="myroot" />
    </wap:adf-web-config>
</adf-settings>
```

A.12 web.xml

Oracle ADF has specific configuration settings for the standard web.xml deployment descriptor file.

When you create a project in JDeveloper that uses JSF technology, a starter web.xml file with default settings is created for you in the /WEB-INF folder. To edit the file, double-click **web.xml** in the Application Navigator to open it in the XML editor.

The following must be configured in web.xml for all applications that use JSF and ADF Faces:

- **JSF servlet and mapping:** The servlet javax.faces.webapp.FacesServlet that manages the request-processing lifecycle for web applications utilizing JSF to construct the user interface.
- **ADF Faces filter and mapping:** A servlet filter to ensure that ADF Faces is properly initialized by establishing a AdfFacesContext object. This filter also processes file uploads.
- **ADF resource servlet and mapping:** A servlet to serve up web application resources (images, style sheets, JavaScript libraries) by delegating to a ResourceLoader.

The JSF servlet and mapping configuration settings are automatically added to the starter web.xml file when you first create a JSF project. When you insert an ADF Faces component into a JSF page for the first time, JDeveloper automatically inserts the configuration settings for ADF Faces filter and mapping, and resource servlet and mapping.

For more information, see the "Configuration in web.xml" section in the *Oracle Fusion Middleware Web User Interface Developer's Guide for Oracle Application Development Framework*.

A.13 logging.xml

ADF Faces leverages the Java Logging API (`java.util.logging.Logger`) to provide logging functionality when you run a debugging session. Java Logging is a standard API that is available in the Java Platform, starting with JDK 1.4. For the key elements, see the section "Java Logging Overview" at <http://java.sun.com/j2se/1.4.2/docs/guide/util/logging/overview.html>.

The `logging.xml` configuration file can be used to configure the following:

- The logging level for Oracle ADF packages.
- Log output location. In addition to the default Log window in JDeveloper, you can redirect the log output to a file.
- The directory path that determines where your log file resides.

For more information, see [Section 29.4, "Using the ADF Logger and Business Component Browser"](#).

B

Oracle ADF Binding Properties

This appendix provides a reference for the properties of the ADF bindings.

This appendix contains the following section:

- [Section B.1, "EL Properties of Oracle ADF Bindings"](#)

B.1 EL Properties of Oracle ADF Bindings

[Table B-1](#) shows the properties that you can use in EL expressions to access values of the ADF binding objects at runtime. The properties appear in alphabetical order.

Table B-1 EL Properties of Oracle ADF Bindings

Runtime Property	Description	Iterator	Action	Attribute	Button	List	Table	Tree
actionEnabled	Use operationEnabled instead.	n/a	yes	n/a	n/a	n/a	n/a	n/a
allRowsInRange	Returns an array of current set of rows from the associated collection. Calls getAllRowsInRange() on the RowSetIterator.	yes	n/a	n/a	n/a	n/a	n/a	n/a
attributeDef	Returns the attribute definition for the first attribute to which the binding is associated.	n/a	n/a	yes	yes	yes	n/a	n/a
attributeDefs	Returns the attribute definitions for all the attributes to which the binding is associated.	n/a	n/a	yes	yes	yes	n/a	n/a
attributeValue	Returns an unformatted and typed (appropriate Java type) value in the current row, for the attribute to which the control binding is bound. Note this property is not visible in the EL expression builder dialog.	n/a	n/a	yes	yes	yes	n/a	n/a

Table B-1 (Cont.) EL Properties of Oracle ADF Bindings

Runtime Property	Description	Iterator	Action	Attribute	Button	List	Table	Tree
attributeValues	Returns the value of all the attributes to which the binding is associated in an ordered array.Returns an array of an unformatted and typed (appropriate Java type) values in the current row for all the attributes to which the control binding is bound. Note this property is not visible in the EL expression builder dialog.	n/a	n/a	yes	yes	yes	n/a	n/a
bindings	Returns a new binding for each cell or attribute exposed under the rows of a tree node binding.	no	no	no	no	no	no	yes
children	Returns the child nodes of a tree node binding.	n/a	n/a	n/a	n/a	n/a	n/a	yes
currentRow	Returns the current row on an action binding bound to an iterator (for example, built-in navigation actions).	n/a	yes	n/a	n/a	n/a	n/a	n/a
dataControl	Returns the iterator's associated data provider.	yes	n/a	n/a	n/a	n/a	n/a	n/a
displayData	Returns a list of map elements. Each map entry contains the following elements: <ul style="list-style-type: none"> ■ selected: A boolean true if current entry should be selected. ■ index: The index value of the current entry. ■ prompt: A string value that may be used to render the entry in the UI. ■ displayValues: An ordered list of display attribute values for all display attributes in the list binding. Note this property is not visible in the EL expression builder dialog.	n/a	n/a	n/a	n/a	yes	n/a	n/a
displayHint	Returns the display hint for the first attribute to which the binding is associated. The hint identifies whether the attribute should be displayed or not. For more information, see <code>oracle.jbo.AttributeHints.displayHint</code> . Note this property is not visible in the EL expression builder dialog.	n/a	n/a	n/a	n/a	yes	n/a	n/a

Table B-1 (Cont.) EL Properties of Oracle ADF Bindings

Runtime Property	Description	Iterator	Action	Attribute	Button	List	Table	Tree
displayHints	Returns a list of name-value pairs for UI hints for all display attributes to which the binding is associated. The map contains the following elements: <ul style="list-style-type: none"> ■ label: The label to display for the current attribute. ■ tooltip: The tooltip to display for the current attribute. ■ displayHint: The display hint for the current attribute. ■ displayHeight: The height in lines for the current attribute. ■ displayWidth: The width in characters for the current attribute. ■ controlType: The control type hint for the current attribute. ■ format: The format to be used for the current attribute. <p>Note this property is not visible in the EL expression builder dialog.</p>	n/a	n/a	n/a	yes	yes	n/a	n/a
enabled	Use operationEnabled.	n/a	n/a	n/a	n/a	n/a	n/a	n/a
enabledString	Returns disabled if the action binding is not ready to be invoked. Otherwise, returns " ".	n/a	yes	n/a	n/a	n/a	n/a	n/a
error	Returns any exception that was cached while updating the associated attribute value for a value binding or when invoking an operation bound by an operation binding.	yes	yes	yes	yes	yes	yes	yes
estimatedRowCount	Returns the maximum row count of the rows in the collection with which this iterator binding is associated	yes	n/a	n/a	n/a	n/a	yes	yes
findMode	Return true if the iterator is currently operating in find mode. Otherwise, returns false.	yes	n/a	n/a	n/a	n/a	n/a	n/a
fullName	Returns the fully qualified name of the binding object in the Oracle ADF binding context.	yes	yes	yes	yes	yes	yes	yes

Table B-1 (Cont.) EL Properties of Oracle ADF Bindings

Runtime Property	Description	Iterator	Action	Attribute	Button	List	Table	Tree
hints	Returns the value of the UI hint indicated for the binding. See displayHints for the list of UI hint keywords.	yes	yes	yes	yes	yes	yes	yes
inputValue	Returns the value of the first attribute to which the binding is associated. If the binding was used to set the value on the attribute and the set operation failed, this method returns the invalid value that was being set.	n/a	n/a	yes	yes	yes	yes	yes
iteratorBinding	Returns the iterator binding that provides access to the data collection.	n/a	yes	yes	yes	yes	yes	yes
label	Returns the label (if supplied by Control Hints) for the first attribute of the binding.	n/a	n/a	yes	yes	yes	n/a	n/a
labels	Returns a map of labels (if supplied by Control Hints) keyed by attribute name for all attributes to which the binding is associated. Note this property is not visible in the EL expression builder dialog.	n/a	n/a	yes	yes	yes	yes	n/a
labelSet	Returns an ordered set of labels for all the attributes to which the binding is associated. Note this property is not visible in the EL expression builder dialog.	n/a	n/a	yes	yes	yes	yes	n/a
mandatory	Returns whether the first attribute to which the binding is associated is required.	n/a	n/a	yes	yes	yes	n/a	n/a
name	Returns the name of the binding object in the context of the binding container to which it is registered. Note this property is not visible in the EL expression builder dialog.	yes	yes	yes	yes	yes	yes	yes
operationEnabled	Returns true or false depending on the state of the action binding. For example, the action binding may be enabled (true) or disabled (false) based on the currency (as determined, for example, when the user clicks the First, Next, Previous, Last navigation buttons).	n/a	yes	n/a	n/a	n/a	n/a	n/a

Table B-1 (Cont.) EL Properties of Oracle ADF Bindings

Runtime Property	Description	Iterator	Action	Attribute	Button	List	Table	Tree
rangeSet	<p>Returns a list of map elements over the range of rows from the associated iterator binding. The elements in this list are wrapper objects over the indexed row in the range that restricts access to only the attributes to which the binding is bound. The properties returned on the reference object are:</p> <ul style="list-style-type: none"> ■ <code>index</code> — The range index of the row this reference is pointing to. ■ <code>key</code> — The key of the row this reference is pointing to. ■ <code>keyStr</code> — The String format of the key of the row this reference is pointing to. ■ <code>currencyString</code> — The current indexed row as a String. Returns "*" if the current entry belongs to the current row; otherwise, returns " ". This property is useful in JSP applications to display the current row. ■ <code>attributeValues</code> — The array of applicable attribute values from the row. <p>And you may also access an attribute value by name on a range set like <code>rangeSet.dname</code> if <code>dname</code> is a bound attribute in the range binding.</p>	n/a	n/a	n/a	n/a	n/a	yes	yes
rangeSize	Returns the range size of the ADF iterator binding's row set. This allows you to determine the number of data objects to bind from the data source.	yes	n/a	n/a	n/a	n/a	yes	yes
rangeStart	Returns the absolute index in a collection of the first row in range. See javadoc for <code>oracle.jbo.RowSetIterator.getRangeStart()</code> .	yes	n/a	n/a	n/a	n/a	yes	yes
result	Returns the result of a method that is bound and invoked by a method action binding.	n/a	yes	n/a	n/a	n/a	n/a	n/a
rootNodeBinding	Returns the root node of a tree binding.	n/a	n/a	n/a	n/a	n/a	n/a	yes

Table B-1 (Cont.) EL Properties of Oracle ADF Bindings

Runtime Property	Description	Iterator	Action	Attribute	Button	List	Table	Tree
selectedValue	Returns the value corresponding to the current selected index in the list or button binding.	n/a	n/a	n/a	yes	yes	n/a	n/a
tooltip	Returns the tooltip hint for the first attribute to which the binding is associated.	n/a	n/a	yes	yes	yes	n/a	n/a
updateable	Returns <code>true</code> if the first attribute to which the binding is associated is updateable. Otherwise, returns <code>false</code> .	n/a	n/a	yes	yes	yes	n/a	n/a

C

Oracle ADF Permission Grants

This appendix lists the security-aware ADF components and the actions that their Permission implementation classes define.

This appendix contains the following section:

- [Appendix C.1, "Grantable Actions of ADF Components"](#)

C.1 Grantable Actions of ADF Components

You can create security policies for the ADF components shown in [Table C-1](#) by defining permission information in the overview editor for ADF security policies. Permissions that you add to the policy store, specify the fully-qualified permission class name, the fully-qualified resource name, the action that can be performed against the resource, and the application role target of the grant. When you enable ADF security to enforce permission checking, the operations supported by ADF components will be inaccessible to users who do not possess sufficient access rights as defined by grants to their application role.

Note: For complete details about defining ADF security policies, see [Chapter 28, "Adding Security to a Fusion Web Application"](#).

Table C-1 Oracle ADF Security Permissions Grants

ADF Component	Grantable Action	Corresponding Implementation
ADF Bounded Task Flow	View	Read and execute a bounded task flow. This is the only operation that the task flow supports in this release.
ADF page definition	View	View the page. This is the only operation that the page definition supports in this release.
ADF Business Components entity objects	read update removeCurrentRow /delete	View a row of the bound collection. Update any attribute of the bound collection. Delete a row from the bound collection.
ADF Business Component attributes of entity objects	update	Update a specific attribute of the bound collection.

D

ADF Equivalents of Common Oracle Forms Triggers

This appendix provides a quick summary of how basic tasks performed with the most common Oracle Forms triggers are accomplished using Oracle ADF.

This appendix contains the following sections:

- [Section D.1, "Validation & Defaulting \(Business Logic\)"](#)
- [Section D.2, "Query Processing"](#)
- [Section D.3, "Database Connection"](#)
- [Section D.4, "Transaction "Post" Processing \(Record Cache\)"](#)
- [Section D.5, "Error Handling"](#)

D.1 Validation & Defaulting (Business Logic)

Table D–1 ADF Equivalents for Oracle Forms Validation and Defaulting Triggers

Forms Trigger	ADF Equivalent
WHEN-VALIDATE-RECORD Execute validation code at the record level	In the custom EntityImpl class for your entity object, write a public method returning a boolean type with a method name like validateXXXX() and have it return true if the validation succeeds or false if the validation fails. Then, add a Method validator for this validation method to your entity object at the entity level. When doing that, you can associate a validation failure message with the rule.
WHEN-VALIDATE-ITEM Execute validation code at the field level	In the custom EntityImpl class for your entity object, write a public method returning a boolean type and accepting a single argument of the same data type as your attribute, having a method name like validateXXXX(). Have it return true if the validation succeeds or false if the validation fails. Then, add a Method validator for this validation method to the entity object at the attribute level for the appropriate attribute. When doing that, you can associate a validation failure message with the rule.
WHEN-DATABASE-RECORD Execute code when a row in the data block is marked for insert or update	Override the addToTransactionManager() method of your entity object. Write code after calling the super.

Table D-1 (Cont.) ADF Equivalents for Oracle Forms Validation and Defaulting Triggers

Forms Trigger	ADF Equivalent
WHEN-CREATE-RECORD Execute code to populate complex default values when a new record in the data block is created, without changing the modification status of the record	Override the <code>create()</code> method of your entity object and after calling the super, use appropriate <code>setAttrName()</code> methods to set default values for attributes as necessary. To immediately set a primary key attribute to the value of a sequence, construct an instance of the <code>SequenceImpl</code> helper class and call its <code>getSequenceNumber()</code> method to get the next sequence number. Assign this value to your primary key attribute.
WHEN-REMOVE-RECORD Execute code whenever a row is removed from the data block	If you want to wait to assign the sequence number until the new record is saved, but still without using a database trigger, you can use this technique in an overridden <code>prepareForDML()</code> method in your entity object. If instead you want to assign the primary key from a sequence using your own BEFOREINSERTFOREACHROW database trigger, then use the special data type called <code>DBSequence</code> for your primary key attribute instead of the regular <code>Number</code> type.
	Override the <code>remove()</code> method of your entity object and write code either before or after calling the super.

D.2 Query Processing

Table D-2 ADF Equivalents for Oracle Forms Query Processing Triggers

Forms Trigger	ADF Equivalent
PRE-QUERY Execute logic before executing a query in a data block, typically to set up values for Query-by-Example criteria in the "example record"	Override the <code>executeQueryForCollection()</code> method on your view object class and write code before calling the super.
ON-COUNT Override default behavior to count the query hits for a data block	Override the <code>getQueryHitCount()</code> method in your view object and do something instead of calling the super.
POST-QUERY Execute logic after retrieving each row from the data source for a data block.	Generally instead of using a POST-QUERY style technique to fetch descriptions from other tables based on foreign key values in the current row, in ADF it's more efficient to build a view object that has multiple participating entity objects, joining in all the information you need in the query from the main table, as well as any auxiliary/lookup-value tables. This way, in a single roundtrip to the database you get all the information you need. If you still need a per-fetched-row trigger like POST-QUERY, override the <code>createInstanceFromResultSet()</code> method in your view object class.
ON-LOCK Override default behavior to attempt to acquire a lock on the current row in the data block	Override the <code>lock()</code> method in your entity object class and do something instead of calling the super.

D.3 Database Connection

Table D–3 ADF Equivalents for Oracle Forms Database Connection Triggers

Forms Trigger	ADF Equivalent
POST-LOGON Execute logic after logging into the database	Override the <code>afterConnect()</code> method on your custom application module. Since application module instances can stay connected while serving different logical client sessions, you can override the <code>prepareSession()</code> method, which is fired after initial login, as well as after any time the application module is accessed by a user that was different from the one that accessed it last time.
PRE-LOGOUT Execute logic before logging out of the database	Override the <code>beforeDisconnect()</code> method on your custom application module class.

D.4 Transaction "Post" Processing (Record Cache)

Table D–4 ADF Equivalents for Oracle Forms Transactional Triggers

Forms Trigger	ADF Equivalent
PRE-COMMIT Execute code before commencing processing of the changed rows in all data blocks in the transaction	Override the <code>commit()</code> method in a custom <code>DBTransactionImpl</code> class and write code before calling the super. Note: For an overview of creating and using a custom <code>DBTransaction</code> implementation, see Section 33.8.4.1, "Creating a Custom Database Transaction Framework Extension Class" .
PRE-INSERT Execute code before a new row in the data block is inserted into the database during "post" processing	Override the <code>doDML()</code> method in your entity class, and if the operation equals <code>DML_INSERT</code> , then write code <i>before</i> calling the super.
ON-INSERT Override default processing for inserting a new row into the database during "post" processing	Override the <code>doDML()</code> method in your entity class, and if the operation equals <code>DML_INSERT</code> , then write code <i>instead of</i> calling the super.
POST-INSERT Execute code after new row in the data block is inserted into the database during "post" processing	Override the <code>doDML()</code> method in your entity class, and if the operation equals <code>DML_INSERT</code> , then write code <i>after</i> calling the super.
PRE-DELETE Execute code before a row removed from the data block is deleted from the database during "post" processing	Override the <code>doDML()</code> method in your entity class, and if the operation equals <code>DML_DELETE</code> , then write code <i>before</i> calling the super.
ON-DELETE Override default processing for deleting a row removed from the data block from the database during "post" processing	Override the <code>doDML()</code> method in your entity class, and if the operation equals <code>DML_DELETE</code> , then write code <i>instead of</i> calling the super.

Table D–4 (Cont.) ADF Equivalents for Oracle Forms Transactional Triggers

Forms Trigger	ADF Equivalent
POST-DELETE Execute code after a row removed from the data block is deleted from the database during "post" processing	Override the <code>doDML()</code> method in your entity class, and if the <code>operation</code> equals <code>DML_DELETE</code> , then write code <i>after</i> calling the super.
PRE-UPDATE Execute code before a row changed in the data block is updated in the database during "post" processing	Override the <code>doDML()</code> method in your entity class, and if the <code>operation</code> equals <code>DML_UPDATE</code> , then write code <i>before</i> calling the super.
ON-UPDATE Override default processing for updating a row changed in the data block from the database during "post" processing	Override the <code>doDML()</code> method in your entity class, and if the <code>operation</code> equals <code>DML_UPDATE</code> , then write code <i>instead of</i> calling the super.
POST-UPDATE Execute code after a row changed in the data block is updated in the database during "post" processing	Override the <code>doDML()</code> method in your entity class, and if the <code>operation</code> equals <code>DML_UPDATE</code> , then write code <i>after</i> calling the super.
POST-FORMS-COMMIT Execute code after Forms has "posted" all necessary rows to the database, but before issuing the data commit to end the transaction	If you want a single block of code for the whole transaction, you can override the <code>doCommit()</code> method in a custom <code>DBTransactionImpl</code> object and write code before calling the super. To execute entity-specific code before commit for each affected entity in the transaction, override the <code>beforeCommit()</code> method on your entity object, and write code there.
POST-DATABASE-COMMIT Execute code after database transaction has been committed	Override the <code>commit()</code> method in a custom <code>DBTransactionImpl</code> class, and write code <i>after</i> calling the super.

D.5 Error Handling

Table D–5 ADF Equivalents for Oracle Forms Error Handling Triggers

Forms Trigger	ADF Equivalent
ON-ERROR Override default behavior for handling an error	Install a custom error handler (<code>DCErrorHandler</code>) on the ADF <code>BindingContext</code> . For details, see Section 26.8, "Customizing Error Handling" .

Most Commonly Used ADF Business Components Methods

This appendix highlights the most commonly used methods in the interfaces and classes of the ADF Business Components layer of Oracle ADF.

This appendix contains the following sections:

- [Section E, "Most Commonly Used ADF Business Components Methods"](#)
- [Section E.2, "Most Commonly Used Methods In the Business Service Tier"](#)

E.1 Most Commonly Used Methods in the Client Tier

All of the interfaces described in this section are designed for use by client-layer code and are part of the `oracle.jbo.*` package.

Note: The corresponding implementation classes for these `oracle.jbo.*` interfaces are consciously designed to *not* be directly accessed by client code. As you'll see in the [Section E.2, "Most Commonly Used Methods In the Business Service Tier"](#) section below, the implementation classes live in the `oracle.jbo.server.*` package and generally have the suffix `Impl` in their name to help remind you not to using them in your client-layer code.

E.1.1 ApplicationModule Interface

The ApplicationModule is a business service component that acts as a transactional container for other ADF components and coordinates *with* them to implement a number of Java EE design patterns important to business application developers. These design pattern implementations enable your client code to work easily with updatable collections of value objects, based on fast-lane reader SQL queries that retrieve only the data needed by the client, in the way the client wants to view it. Changes made to these value objects are automatically coordinated with your persistent business domain objects in the business service tier to enforce business rules consistently and save changes back to the database.

Note: For the complete list of design patterns ADF Business Components implements, see [Appendix F, "ADF Business Components Java EE Design Pattern Catalog"](#).

Table E-1 ApplicationModule Interface

If you want to...	Call this ApplicationModule interface method...
Access an existing view object instance using the assigned instance name (for example, MyVOInstanceName)	<code>findViewObject()</code>
Creating a new view object instance from an existing definition	<code>createViewObject()</code>
Creating a new view object instance from a SQL Statement	<code>createViewObjectFromQueryStmt()</code> Note: This incurs runtime overhead to describe the "shape" of the dynamic query's SELECT list. Oracle recommends using this only when you cannot know the SELECT list for the query at design-time. Furthermore, if you are creating the dynamic query based on some kind of custom runtime repository, you can follow the steps to create (both read-only and updatable) dynamic view objects without the runtime-describe overhead, as described in Section 35.10, "Creating a View Object with Multiple Updatable Entities" . If only the WHERE needs to be dynamic, create the view object at design time, then set the where clause dynamically as needed using <code>ViewObject</code> API's.
Access a nested application module instance by name	<code>findApplicationModule()</code>
Create a new nested application module instance from an existing definition	<code>createApplicationModule()</code>
Find a view object instance in a nested application module using a dot-notated name (for example, MyNestedAMInstanceName.OneOfItsVONames)	<code>findViewObject()</code> Note: You can use the above method to find an instance of a view object belonging to a nested application module in a single method call. This way you do not need to first call <code>findApplicationModule()</code> to find the nested application module, then call <code>findViewObject()</code> on that nested application module.
Accessing the current transaction object	<code>getTransaction()</code>

In addition to generic `ApplicationModule` access, Oracle JDeveloper can generate you a custom `YourApplicationModuleName` interface containing service-level custom methods that you've chosen to expose to the client. You do this by visiting the **Client Interface** tab of the Application Module editor, and shuttling the methods you'd like to appear in your client interface into the **Selected** list.

E.1.2 Transaction Interface

The Transaction interface exposes methods allowing the client to manage pending changes in the current transaction.

Table E–2 Transaction Interface

If you want to...	Call this Transaction interface method...
Commit pending changes	<code>commit()</code>
Rollback pending changes	<code>rollback()</code>
Execute a one-time database command or block of PL/SQL	<code>executeCommand()</code>
	Note: Commands that require retrieving OUT parameters, that will be executed more than once, or that could benefit by using bind variables should not use this method. Instead, expose a custom method on your application module.
Validate all pending invalid changes in the transaction	<code>validate()</code>
Change the default locking mode	<code>setLockingMode()</code>
	Note: You can set the locking mode in your configuration by setting the property <code>jbo.locking.mode</code> to one of the four supported values: <code>none</code> , <code>optimistic</code> , <code>pessimistic</code> , <code>optupdate</code> . If you don't explicitly set it, it will default to <code>pessimistic</code> . For Fusion web applications, Oracle recommends using <code>optimistic</code> or <code>optupdate</code> modes.
Decide whether to use bundled exception reporting mode or not.	<code>setBundledExceptionMode()</code>
	Note: ADF controller layer support sets this parameter to <code>true</code> automatically for Fusion web applications.
Decide whether entity caches will be cleared upon a successful commit of the transaction.	<code>setClearCacheOnCommit()</code>
	Note: Default is <code>false</code>
Decide whether entity caches will be cleared upon a rollback of the transaction.	<code>setClearCacheOnRollback()</code>
	Note: Default is <code>true</code>
Clear the entity cache for a specific entity object.	<code>clearEntityCache()</code>

E.1.3 ViewObject Interface

A ViewObject encapsulates a database query and simplifies working with the RowSet of results it produces. You use view objects to project, filter, join, or sort business data using SQL from one or more tables into exactly the format that the user should see it on the page or panel. You can create "master/detail" hierarchies of any level of depth or complexity by connecting view objects together using view links. View objects can produce read-only query results, or by associating them with one or more entity objects at design time, can be fully updatable. Updatable view objects can support insertion, modification, and deletion of rows in the result collection, with automatic delegation to the correct business domain objects.

Every `ViewObject` aggregates a "default rowset" for simplifying the 90% of use cases where you work with a single `RowSet` of results for the `ViewObject`'s query. A `ViewObject` implements all the methods on the `RowSet` interface by delegating them to this default `RowSet`. That means you can invoke any `RowSet` methods on any `ViewObject` as well.

Every `ViewObject` implements the `StructureDef` interface to provide information about the number and types of attributes in a row of its row sets. So you can call `StructureDef` methods right on any view object.

Table E-3 ViewObject Interface

If you want to...	Call this <code>ViewObject</code> interface method...
Set an additional runtime WHERE clause on the rowset	<code>setWhereClause()</code> Note: This WHERE clause augments any WHERE clause specified at design time in the base view object. It does not replace it.
Set a dynamic ORDER BY clause	<code>setOrderByClause()</code>
Create a Query-by-Example criteria collection	<code>createViewCriteria()</code> Note: You then create one or more <code>ViewCriteriaRow</code> objects using the <code>createViewCriteriaRow()</code> method on the <code>ViewCriteria</code> object you created. Then, you <code>add()</code> these view criteria rows to the view criteria collection and apply the criteria using the method below.
Apply a Query-by-Example criteria collection	<code>applyViewCriteria()</code>
Set a query optimizer hint	<code>setQueryOptimizerHint()</code>
Access the attribute definitions for the key attributes in the view object	<code>getKeyAttributeDefs()</code>
Add a dynamic attribute to rows in this view object's row sets	<code>addDynamicAttribute()</code>
Clear all row sets produced by a view object	<code>clearCache()</code>
Remove view object instance and its resources	<code>remove()</code>
Set an upper limit on the number of rows that the view object will attempt to fetch from the database.	<code>setMaxFetchSize()</code> Note: Default is -1 which means to impose no limit on how many rows would be retrieved from the database if you iterate through them all. By default they are fetched lazily as you iterate through them.

In addition to generic `ViewObject` access, JDeveloper can generate you a custom `YourViewObjectName` interface containing view-object level custom methods that you've chosen to expose to the client. You do this by visiting the **Client Interface** tab of the View Object editor, and shuttling the methods you'd like to appear in your client interface into the **Selected** list.

E.1.4 RowSet Interface

A `RowSet` is a set of rows, typically produced by executing a `ViewObject`'s query.

Every RowSet aggregates a "default rowset iterator" for simplifying the 90% of use cases where you only need a single iterator over the rowset. A RowSet implements all the methods on the RowSetIterator interface by delegating them to this default RowSetIterator. This means you can invoke any RowSetIterator method on any RowSet (or ViewObject, since it implements RowSet as well for its default RowSet).

Table E–4 RowSet Interface

If you want to...	Call this RowSet interface method...
Set a where clause bind variable value	<code>setWhereClauseParam()</code>
	Note: Bind variable ordinal positions are zero-based
Avoid view object row caching if data is being read only once	<code>setForwardOnly()</code>
Force a row set's query to be (re)executed	<code>executeQuery()</code>
Estimate the number of rows in a view object's query result	<code>getEstimatedRowCount()</code>
Produce XML document for rows in View Object rowset	<code>writeXML()</code>
Process all rows from an incoming XML document	<code>readXML()</code>
Set whether rowset will automatically see new rows based on the same entity object created through other rowsets	<code>setAssociationConsistent()</code>
Create secondary iterator to use for programmatic iteration	<code>createRowSetIterator()</code>
	Note: If you plan to find and use the secondary iterator by name later, then pass in a string name as the argument, otherwise pass null for the name and make sure to close the iterator when done iterating by calling its <code>closeRowSetIterator()</code> method.

E.1.5 RowSetIterator Interface

A RowSetIterator is an iterator over the rows in a RowSet. By default it allows you to iterate both forward and backward through the rows.

Table E–5 RowSetIterator Interface

If you want to...	Call this RowSetIterator interface method...
Get the first row of the iterator's rowset	<code>first()</code>
Test whether there are more rows to iterate	<code>hasNext()</code>
Get the next row of iterator's rowset	<code>next()</code>
Find row in this iterator's rowset with a given Key value	<code>findByKey()</code>
	Note: It's important that the Key object that you pass to <code>findByKey</code> be created using the <i>exact</i> same datatypes as the attributes that comprise the key of the rows in the view object you're working with.

Table E–5 (Cont.) RowSetIterator Interface

If you want to...	Call this RowSetIterator interface method...
Create a new row to populate for insertion	<p><code>createRow()</code></p> <p>Note:</p> <p>The new row will already have default values set for attributes which either have a static default value supplied at the entity object or view object level, or if the values have been populated in an overridden <code>create()</code> method of the underlying entity object(s).</p>
Create a view row with an initial set of foreign key and/or discriminator attribute values	<p><code>createAndInitRow()</code></p> <p>Note:</p> <p>You use this method when working with view objects that can return one of a "family" of entity object subtypes. By passing in the correct discriminator attribute value in the call to create the row, the framework can create you the correct matching entity object subtype underneath.</p>
Insert a new row into the iterator's rowset	<p><code>insertRow()</code></p> <p>Note:</p> <p>It's a good habit to always immediately insert a newly created row into the rowset. That way you will avoid a common gotcha of creating the row but forgetting to insert it into the rowset.</p>
Get the last row of the iterator's rowset	<code>last()</code>
Get the previous row of the iterator's rowset	<code>previous()</code>
Reset the current row pointer to the slot before the first row	<code>reset()</code>
Close an iterator when done iterating	<code>closeRowSetIterator()</code>
Set a given row to be the current row	<code>setCurrentRow()</code>
Remove the current row	<code>removeCurrentRow()</code>
Remove the current row to later insert it at a different location in the same iterator.	<code>removeCurrentRowAndRetain()</code>
Remove the current row from the current collection but do not remove it from the transaction.	<code>removeCurrentRowFromCollection()</code>
Set/change the number of rows in the range (a "page" of rows the user can see)	<code>setRangeSize()</code>
Scroll to view the Nth page of rows (1-based)	<code>scrollToRangePage()</code>
Scroll to view the range of rows starting with row number N	<code>scrollRangeTo()</code>
Set row number N in the range to be the current row	<code>setCurrentRowAtRangeIndex()</code>
Get all rows in the range as a Row array	<code>getAllRowsInRange()</code>

E.1.6 Row Interface

A Row is generic value object. It contains attributes appropriate in name and Java type for the ViewObject that it's related to.

Table E–6 Row Interface

If you want to...	Call this Row interface method...
Get the value of an attribute by name	getAttribute()
Set the value of an attribute by name	setAttribute()
Produce an XML document for a single row	writeXML()
Eagerly validate a row	validate()
Read row attribute values from XML	readXML()
Remove the row	remove()
Flag a newly created row as temporary (until updated again)	setNewRowState(Row.STATUS_INITIALIZED)
Retrieve the attribute structure definition information for a row	getStructureDef()
Get the Key object for a row	getKey()

In addition to generic Row access, JDeveloper can generate you a custom *YourViewObjectNameRow* interface containing your type-safe attribute getter and setter methods, as well as any desired row-level custom methods that you've chosen to expose to the client. You do this by visiting the **Client Row Interface** tab of the View Object editor, and shuttling the methods you'd like to appear in your client interface into the **Selected** list.

E.1.7 StructureDef Interface

A StructureDef is an interface that provides access to runtime metadata about the structure of a Row.

In addition, for convenience every ViewObject implements the StructureDef interface as well, providing access to metadata about the attributes in the resulting view rows that its query will produce.

Table E–7 StructureDef Interface

If you want to...	Call this StructureDef interface method...
Access attribute definitions for all attributes in the view object row	getAttributeDefs()
Find an attribute definition by name	findAttributeDef()
Get attribute definition by index	getattributeDef()
Get number of attributes in a row	getAttributeCount()

E.1.8 AttributeDef Interface

An AttributeDef provides attribute definition information for any attribute of a View Object row or Entity Object instance like attribute name, Java type, and SQL type. It also provides access to custom attribute-specific metadata properties that can be inspected by generic code you write, as well as UI hints that can assist in rendering an appropriate user interface display for the attribute and its value.

Table E–8 AttributeDef Interface

If you want to...	Call this AttributeDef interface method...
Get the Java type of the attribute	<code>getJavaType()</code>
Get the SQL type of the attribute	<code>getSQLType()</code>
	Note: The int value corresponds to constants in the JDBC class <code>java.sql.Types</code>
Determine the kind of attribute	<code>getAttributeKind()</code>
	Note: If it's a simple attribute, it returns one of the constants <code>ATTR_PERSISTENT</code> , <code>ATTR_SQL_DERIVED</code> , <code>ATTR_TRANSIENT</code> , <code>ATTR_DYNAMIC</code> , <code>ATTR_ENTITY_DERIVED</code> . If it is an 1-to-1 or many-to-1 association/viewlink accessor it returns <code>ATTR_ASSOCIATED_ROW</code> . If it is an 1-to-many or many-to-many association/viewlink accessor it returns <code>ATTR_ASSOCIATED_ROWITERATOR</code>
Get the Java type of elements contained in an Array-valued attribute	<code>getElemJavaType()</code>
Get the SQL type of elements contained in an Array-valued attribute	<code>getElemSQLType()</code>
Get the name of the attribute	<code>getName()</code>
Get the index position of the attribute	<code>getIndex()</code>
Get the precision of a numeric attribute or the maximum length of a String attribute	<code>getPrecision()</code>
Get the scale of a numeric attribute	<code>getScale()</code>
Get the underlying column name corresponding to the attribute	<code>getColumnNameForQuery()</code>
Get attribute-specific custom property values	<code>getProperty()</code> , <code>getProperties()</code>
Get the UI AttributeHints object for the attribute	<code>getUIHelper()</code>
Test whether the attribute is mandatory	<code>isMandatory()</code>
Test whether the attribute is queriable	<code>isQueriable()</code>
Test whether the attribute is part of the primary key for the row	<code>isPrimaryKey()</code>

E.1.9 AttributeHints Interface

The AttributeHints interface related to an attribute exposes UI hint information that attribute that you can use to render an appropriate user interface display for the attribute and its value.

Table E–9 AttributeHints Interface

If you want to...	Call this AttributeHints interface method...
Get the UI label for the attribute	<code>getLabel()</code>
Get the tool tip for the attribute	<code>getTooltip()</code>

Table E-9 (Cont.) AttributeHints Interface

If you want to...	Call this AttributeHints interface method...
Get the formatted value of the attribute, using any format mask supplied	getFormattedAttribute()
Get the display hint for the attribute	getDisplayHint()
	Note: Will have a String value of either Display or Hide.
Get the preferred control type for the attribute	getControlType()
Parse a formatted string value using any format mask supplied for the attribute	parseFormattedAttribute()

E.2 Most Commonly Used Methods In the Business Service Tier

The implementation classes corresponding to the `oracle.jbo.*` interfaces, as described in [Section E.1, "Most Commonly Used Methods in the Client Tier"](#), are consciously designed to *not* be directly accessed by client code. They live in a different package named `oracle.jbo.server.*` and have the `Impl` suffix in their name to help remind you not to use them in your client-layer code.

In your business service tier implementation code, you can use any of the same methods that are available to clients, but in addition you can also:

- Safely cast any `oracle.jbo.*` interface to its `oracle.jbo.server.*` package implementation class and use any methods on that `Impl` class as well.
- Override any of the base framework implementation class' public or protected methods to augment or change its default functionality by writing custom code in your component subclass before or after calling `super.methodName()`.

This section provides a summary of the most frequently called, written, and overridden methods for the key ADF Business Components classes.

E.2.1 Controlling Custom Java Files For Your Components

Before examining the specifics of individual classes, it's important to understand how you can control which custom Java files each of your components will use. When you don't need a customized subclass for a given component, you can just let the base framework class handle the implementation at runtime.

Each business component you create comprises a single XML component descriptor, and zero or more related custom Java implementation files. Each component that supports Java customization has a **Java** tab in its component editor in the JDeveloper IDE. By checking or unchecking the different Java classes, you control which ones get created for your component. If none of the boxes is checked, then your component will be an XML-only component, which simply uses the base framework class as its Java implementation. Otherwise, tick the checkbox of the related Java classes for the current component that you need to customize. JDeveloper will create you a custom *subclass* of the framework base class in which you can add your code.

Note: You can setup global IDE preferences for which Java classes should be generated by default for each ADF business component type by selecting **Tools | Preferences... | Business Components** and ticking the checkboxes to indicate what you want your defaults to be.

A best practice is to *always* generate Entity Object and View Row classes, even if you don't require any custom code in them other than the automatically-generated getter and setter methods. These getter and setter methods offer you compile-time type checking that avoids discovering errors at runtime when you accidentally set an attribute to an incorrect kind of value.

E.2.2 ApplicationModuleImpl Class

The ApplicationModuleImpl class is the base class for application module components. Since the application module is the ADF component used to implement a business service, think of the application module class as the place where you can write your service-level application logic. The application module coordinates with view object instances to support updatable collections of value objects that are automatically "wired" to business domain objects. The business domain objects are implemented as ADF entity objects.

E.2.2.1 Methods You Typically Call on ApplicationModuleImpl

Table E-10 Methods You Typically Call on ApplicationModuleImpl

If you want to...	Call this method of the ApplicationModuleImpl class
Perform any of the common application module operations from inside your class, which can also be done from the client	For more information, see Section E.1.1, "ApplicationModule Interface" .
Access a view object instance that you added to the application module's data model at design time	<code>getViewObjectName()</code> Note: JDeveloper generates this type-safe view object instance getter method for you to reflect each view object instance in the application module's design-time data-model.
Access the current DBTransaction object	<code>getDBTransaction()</code>
Access a nested application module instance that you added to the application module at design time	<code>getAppModuleInstanceName()</code> Note: JDeveloper generates this type-safe application module instance getter method for you to reflect each nested application module instance added to the current application module at design time.

E.2.2.2 Methods You Typically Write in Your Custom ApplicationModuleImpl Subclass

Table E–11 Methods You Typically Write in Your Custom ApplicationModuleImpl Subclass

If you want to...	Write a method like this in your custom ApplicationModuleImpl class
Invoke a database stored procedure	<p><code>someCustomMethod()</code></p> <p>Note:</p> <p>Use appropriate method on the <code>DBTransaction</code> interface to create a JDBC <code>PreparedStatement</code>. If the stored procedure has <code>OUT</code> parameters, then create a <code>CallableStatement</code> instead.</p>
Expose custom business service methods on your application module	<p><code>someCustomMethod()</code></p> <p>Note:</p> <p>Select the method name on the Client Interface panel of the application module editor to expose it for client access if required.</p>

JDeveloper can generate you a custom `YourApplicationModuleName` interface containing service-level custom methods that you've chosen to expose to the client. You do this by visiting the **Client Interface** tab of the Application Module editor, and shuttling the methods you'd like to appear in your client interface into the **Selected** list.

E.2.2.3 Methods You Typically Override in Your Custom ApplicationModuleImpl Subclass

Table E-12 Methods You Typically Override in Your Custom ApplicationModuleImpl Subclass

If you want to...	Override this method of the ApplicationModuleImpl class
Perform custom setup code the first time an application module is created and each subsequent time it gets used by a different client session.	<pre>prepareSession()</pre> <p>Note:</p> <p>This is the method you'd use to setup per-client context info for the current user in order to use database Oracle's Virtual Private Database (VPD) features. It can also be used to set other kinds of PL/SQL package global variables, whose values might be client-specific, on which other stored procedures might rely.</p>
Perform custom setup code after the application module's transaction is associated with a database connection from the connection pool.	<pre>afterConnect()</pre> <p>Note:</p> <p>Can be a useful place to write a line of code that uses <code>getDBTransaction().executeCommand()</code> to perform an <code>ALTER SESSION SET SQL TRACE TRUE</code> to enable database SQL Trace logging for the current application connection. These logs can then be processed with the TKPROF utility to study the SQL statements being performed and the query optimizer plans that are getting used.</p> <p>For details about the TKPROF utility, see section "Understanding SQL Trace and TKPROF" in the <i>Oracle Database Performance Tuning Guide</i>.</p>
Perform custom setup code before the application module's transaction releases its database connection back to the database connection pool.	<pre>beforeDisconnect()</pre> <p>Note:</p> <p>If you have set <code>jbo.doconnectionpooling</code> to true, then the connection is released to the database connection pool each time the application module is returned to the application module pool.</p>
Write custom application module state to the state management XML snapshot	<pre>passivateState()</pre>
Read and restore custom application module state from the state management XML snapshot	<pre>activateState()</pre>

E.2.3 DBTransactionImpl2 Class

The `DBTransactionImpl2` class — which extends the base `DBTransactionImpl` class, and is constructed by the `DatabaseTransactionFactory` class — is the base class that

implements the DBTransaction interface, representing the unit of pending work in the current transaction.

E.2.3.1 Methods You Typically Call on DBTransaction

Table E-13 Methods You Typically Call on DBTransaction

If you want to...	Call this method on the DBTransaction object
Commit the transaction	commit()
Rollback the transaction	rollback()
Eagerly validate any pending invalid changes in the transaction	validate()
Create a JDBC PreparedStatement using the transaction's Connection object	createPreparedStatement()
Create a JDBC CallableStatement using the transaction's Connection object	createCallableStatement()
Create a JDBC Statement using the transaction's Connection object	createStatement()
Add a warning to the transaction's warning list.	addWarning()

E.2.3.2 Methods You Typically Override in Your Custom DBTransactionImpl2 Subclass

Table E-14 Methods You Typically Override in Your Custom DBTransactionImpl2 Subclass

If you want to...	Override this method in your custom DBTransactionImpl2 class
Perform custom code before or after the transaction commit operation	commit()
Perform custom code before or after the transaction rollback operation	rollback()

In order to have your custom DBTransactionImpl2 subclass get used at runtime, there are two steps you must follow:

1. Create a custom subclass of DatabaseTransactionFactory that overrides the create method to return an instance of your custom DBTransactionImpl2 subclass like this:

```
package com.yourcompany.adfextensions;
import oracle.jbo.server.DBTransactionImpl2;
import oracle.jbo.server.DatabaseTransactionFactory;
import com.yourcompany.adfextensions.CustomDBTransactionImpl;
public class CustomDatabaseTransactionFactory
    extends DatabaseTransactionFactory {
    /**
     * Return an instance of our custom CustomDBTransactionImpl class
     * instead of the default implementation.
     */
}
```

```

        * @return An instance of our custom DBTransactionImpl2 implementation.
        */
    public DBTransactionImpl2 create() {
        return new CustomDBTransactionImpl();
    }
}

```

2. Tell the framework to use your custom transaction factory class by setting the value of the TransactionFactory configuration property to the fully-qualified class name of your custom transaction factory. As with other configuration properties, if not supplied in the configuration XML file, it can be provided alternatively as a Java system parameter of the same name.

E.2.4 EntityImpl Class

The EntityImpl class is the base class for entity objects, which encapsulate the data, validation rules, and business behavior for your business domain objects.

E.2.4.1 Methods You Typically Call on EntityImpl

Table E-15 Methods You Typically Call on EntityImpl

If you want to...	Call this method in your EntityImpl subclass
Get the value of an attribute	<code>getAttributeName()</code> Note: Code-generated getter method calls <code>getAttributeInternal()</code> but provides compile-time type checking.
Set the value of an attribute	<code>setAttributeName()</code> Note: Code-generated setter method calls <code>setAttributeInternal()</code> but provides compile-time type checking.
Get the value of an attribute by name	<code>getAttributeInternal()</code>
Set the value of an attribute by name	<code>setAttributeInternal()</code>
Eagerly perform entity object validation	<code>validate()</code>
Refresh the entity from the database	<code>refresh()</code>
Populate the value of an attribute without marking it as being changed, but sending notification of its being changed so UI's refresh the value on the screen/page.	<code>populateAttributeAsChanged()</code>
Access the definition object for an entity	<code>getDefinitionObject()</code>
Get the Key object for an entity	<code>getKey()</code>
Determine the state of the entity instance, irrespective of whether it has already been posted in the current transaction (but not yet committed)	<code>getEntityState()</code> Note: Will return one of the constants <code>STATUS_UNMODIFIED</code> , <code>STATUS_INITIALIZED</code> , <code>STATUS_NEW</code> , <code>STATUS_MODIFIED</code> , <code>STATUS_DELETED</code> , or <code>STATUS_DEAD</code> indicating the status of the entity instance in the current transaction.

Table E-15 (Cont.) Methods You Typically Call on EntityImpl

If you want to...	Call this method in your EntityImpl subclass
Determine the state of the entity instance	<code>getPostState()</code> Note: This method is typically only relevant if you are programmatically using the <code>postChanges()</code> method to post but not yet commit entity changes to the database and need to detect the state of an entity with regard to its posting state
Get the value originally read from the database for a given attribute	<code>getPostedAttribute()</code>
Eagerly lock the database row for an entity instance	<code>lock()</code>

E.2.4.2 Methods You Typically Write in Your Custom EntityImpl Subclass

Table E-16 Methods You Typically Write in Your Custom EntityImpl Subclass

If you want to...	Write a method like this in your EntityImpl subclass
Perform attribute-specific validation	<code>public boolean validateSomething(AttrTypevalue)</code> Note: Register the attribute validator method by adding a "MethodValidator" on correct attribute in the Validation panel of the Entity Object editor.
Perform entity-level validation	<code>public boolean validateSomething()</code> Note: Register the entity-level validator method by adding a "MethodValidator" on the entity in the Validation panel of the Entity Object editor.
Calculate the value of a transient attribute	Add your calculation code to the generated <code>getAttributeName()</code> method.

E.2.4.3 Methods You Typically Override on EntityImpl

Table E-17 Methods You Typically Override on EntityImpl

If you want to...	Override this method in your custom EntityImpl subclass...
Set calculated default attribute values, including programmatically populating the primary key attribute value of a new entity instance.	<code>create()</code> Note: After calling <code>super.create()</code> , call the appropriate <code>setAttributeName()</code> method(s) to set the default values for that(/those) attributes.

Table E-17 (Cont.) Methods You Typically Override on EntityImpl

If you want to...	Override this method in your custom EntityImpl subclass...
Modify attribute values before changes are posted to the database	<code>prepareForDML()</code>
Augment/change the standard INSERT, UPDATE, or DELETE DML operation that the framework will perform on your entity object's behalf to the database	<code>doDML()</code> Note: Check the value of the operation flag to the constants <code>DML_INSERT</code> , <code>DML_UPDATE</code> , or <code>DML_DELETE</code> to test what DML operation is being performed.
Perform complex, SQL-based validation after all entity instances have been posted to the database but before those changes are committed.	<code>beforeCommit()</code>
Insure that a related, newly-created, parent entity gets posted to the database <i>before</i> the current child entity on which it depends	<code>postChanges()</code> Note: If the parent entity is related to this child entity via a composition association, then the framework already handles this automatically. If they are only associated (but not composed) then you need to override <code>postChanges()</code> method to force a newly-created parent entity to post before the current, dependent child entity. For an example of the code you typically write in your overridden <code>postChanges()</code> method to accomplish this, see Section 34.8.3, "Overriding postChanges() to Control Post Order" .

Note: It is possible to write attribute-level validation code directly inside the appropriate `setAttributeName` method of your `EntityImpl` class, however adopting the `MethodValidator` approach suggested in [Table E-16](#) results in having a single place on the **Validation Rules** panel of the overview editor for the attributes of the entity object to look in order to understand all of the validations in effect for your entity object.

WARNING: It is also possible to override the `validateEntity()` method to write entity-level validation code, however if you want to maintain the benefits of the ADF bundled exception mode — where the framework collects and reports a *maximal* set of validation errors back to the client user interface — it is recommended to adopt the `MethodValidator` approach suggested in [Table E-16](#). This allows the framework to automatically collect all of your exceptions that your validation methods throw without your having to understand the bundled exception implementation mechanism. Overriding the `validateEntity()` method directly shifts the responsibility on your *own* code to correctly catch and bundle the exceptions like Oracle ADF would have done by default, which is non-trivial and a chore to remember and hand-code each time.

E.2.5 EntityDefImpl Class

The EntityDefImpl class is a singleton, shared metadata object for all entity objects of a given type in a single Java VM. It defines the structure of the entity instances and provides methods to create new entity instances and find existing instances by their primary key.

E.2.5.1 Methods You Typically Call on EntityDefImpl

Table E–18 Methods You Typically Call on EntityDefImpl

If you want to...	Call the EntityDefImpl method
Find an entity object of a this type by its primary key	<code>findByPrimaryKey()</code> Note: For a tip about getting <code>findByPrimaryKey()</code> to find entity instances of subtype entities as well, see Section 34.7.4.2, "Finding Subtype Entities by Primary Key" .
Access the current DBTransaction object	<code>getDBTransaction()</code>
Find any EntityDefImpl object by its fully-qualified name	<code>findDefObject()</code> (static method)
Retrieve the value of an entity object's custom property	<code>getProperty(), getProperties()</code>
Set the value of an entity object's custom property	<code>setProperty()</code>
Create a new instance of an entity object	<code>createInstance2()</code> Note: Alternatively, you can expose custom <code>createXXX()</code> methods with your own expected signatures in that same custom EntityDefImpl subclass. See the next section for details.
Iterate over the entity instances in the cache of this entity type.	<code>getAllEntityInstancesIterator()</code>
Access ArrayList of entity definition objects for entities that extend the current one.	<code>getExtendedDefObjects()</code>

E.2.5.2 Methods You Typically Write on EntityDefImpl

Table E-19 Methods You Typically Write on EntityDefImpl

If you want to...	Write a method like this in your custom EntityDefImpl class
Allow other classes to create an entity instance with an initial type-safe set of attribute values or setup information.	<pre>createXXXX(Type1arg1, ..., TypeNargN)</pre> <p>Note: Internally, this would create and populate an instance of a <code>NameValuePairs</code> object (which implements <code>AttributeList</code>) and call the protected method <code>createInstance()</code>, passing that <code>NameValuePairs</code> object. Make sure the method is <code>public</code> if other classes need to be able to call it.</p>

E.2.5.3 Methods You Typically Override on EntityDefImpl

Table E-20 Methods You Typically Override on EntityDefImpl

If you want to...	Call the EntityDefImpl method
Perform custom metadata initialization when this singleton metaobject is loaded.	<code>createDef()</code>
Avoid using the <code>RETURNING INTO</code> clause to support refresh-on-insert or refresh-on-update attributes	<code>isUseReturningClause()</code> <p>Note: Return false to disable the use of <code>RETURNING INTO</code>, necessary sometimes when your entity object is based on a view with <code>INSTEAD OF</code> triggers that doesn't support <code>RETURNING INTO</code> at the database level.</p>
Control whether the <code>UPDATE</code> statements issued for this entity update only changed columns, or all columns	<code>isUpdateChangedColumns()</code> <p>Note: Defaults to true.</p>
Find any <code>EntityDefImpl</code> object by its fully-qualified name	<code>findDefObject()</code> <p>Note: Static method.</p>
Set the value of an entity object's custom property	<code>setProperty()</code>
Allow other classes to create a new instance an entity object without doing so implicitly via a view object.	<code>createInstance()</code> <p>Note: If you don't write a custom create method as noted in the previous section, you'll need to override this method and widen the visibility from <code>protected</code> to <code>public</code> to allow other classes to construct an entity instance.</p>

E.2.6 ViewObjectImpl Class

The `ViewObjectImpl` class the base class for view objects.

E.2.6.1 Methods You Typically Call on ViewObjectImpl

Table E–21 Methods You Typically Call on ViewObjectImpl

If you want to...	Call this ViewObjectImpl method
Perform any of the common view object, rowset, or rowset iterator operations from inside your class, which can also be done from the client	For more information, see Section E.1.3, "ViewObject Interface" , Section E.1.4, "RowSet Interface" , and Section E.1.5, "RowSetIterator Interface" .
Set an additional runtime WHERE clause on the default rowset	<code>setWhereClause()</code>
Defines a named bind parameter.	<code>defineNamedWhereClauseParam()</code>
Removes a named bind parameter.	<code>removeNamedWhereClauseParam()</code>
Set bind variable values on the default rowset by name. Only works when you have formally defined named bind variables on your view object.	<code>setNamedWhereClauseParam()</code>
Set bind variable values on the default rowset. Use this method for view objects with binding style of "Oracle Positional" or "JDBC Positional" when you have not formally defined named bind variables.	<code>setWhereClauseParam()</code>
Retrieved a subset of rows in a view object's row set based on evaluating an in-memory filter expression.	<code>getFilteredRows()</code>
Retrieved a subset of rows in the current range of a view object's row set based on evaluating an in-memory filter expression.	<code>getFilteredRowsInRange()</code>
Set the number of rows that will be fetched from the database per round-trip for this view object.	<code>setFetchSize()</code> Note: The default fetch size is a single row at a time. This is definitely not optimal if your view object intends to retrieve many rows, so you should either set the fetch size higher at design time on the Tuning tab of the View Object editor, or set it at runtime using this API.

E.2.6.2 Methods You Typically Write in Your Custom ViewObjectImpl Subclass

Table E–22 Methods You Typically Write in Your Custom ViewObjectImpl Subclass

If you want to...	Write a method like this in your ViewObjectImpl subclass
Provide clients with type-safe methods to set bind variable values without exposing positional details of the bind variables themselves	<code>someMethodName(Type1arg1, ..., TypeNargN)</code> Note: Internally, this method would call the <code>setWhereClauseParam()</code> API to set the correct bind variables with the values provided in the type-safe method arguments.

JDeveloper can generate you a custom `YourViewObjectName` interface containing view object custom methods that you've chosen to expose to the client. You can accomplish this by visiting the **Client Interface** tab of the View Object editor, and

shuttling the methods you'd like to appear in your client interface into the **Selected** list.

E.2.6.3 Methods You Typically Override in Your Custom ViewObjectImpl Subclass

Table E–23 Methods You Typically Override in Your Custom ViewObjectImpl Subclass

If you want to...	Override this ViewObjectImpl method
Initialize custom view object class members (not row attributes!) when the view object instance is created for the first time.	<p><code>create()</code></p> <p>Note:</p> <p>This method is useful to perform setup logic that is applicable to every instance of a view object that will ever get created, in the context of any application module.</p> <p>If instead of <i>generic</i> view object setup logic, you need to perform logic specific to a given view object <i>instance</i> in an application module, then override the <code>prepareSession()</code> method of your application module's <code>ApplicationModuleImpl</code> subclass and perform the logic there after calling <code>findViewObject()</code> to find the view object instance whose properties you want to set.</p>
Write custom view object instance state to the state management XML snapshot.	<code>passivateState()</code>
Read and restore custom view object instance state from the state management XML snapshot.	<code>activateState()</code>
Customize the behavior of view object query execution, independent of whether the query was executed explicitly by calling <code>executeQuery()</code> or implicitly, for example, by navigating to the <code>first()</code> row when the query hadn't yet been executed.	<code>executeQueryForCollection()</code>
Change/augment the way that the <code>ViewCriteria</code> collection of <code>ViewCriteriaRows</code> is converted into a Query-by-Example WHERE clause.	<code>getViewCriteriaClause()</code>

E.2.7 ViewRowImpl Class

The `ViewRowImpl` class is the base class for view row objects.

E.2.7.1 Methods You Typically Call on ViewRowImpl

Table E–24 Methods You Typically Call on ViewRowImpl

If you want to...	Write a method like this in your custom ViewRowImpl class
Perform any of the common view row operations from inside your class, which can also be done from the client.	For more information, see Section E.1.6, "Row Interface" .
Get the value of an attribute.	<code>getAttrName()</code>
Set the value of an attribute.	<code>setAttrName()</code>

Table E–24 (Cont.) Methods You Typically Call on ViewRowImpl

If you want to...	Write a method like this in your custom ViewRowImpl class
Access the underlying entity instance to which this view row is delegating attribute storage.	<pre>getEntityUsageAliasName()</pre> <p>Note: You can change the name of the entity usage alias name on the Entity Objects tab of the View Object Editor</p>

E.2.7.2 Methods You Typically Write on ViewRowImpl

Table E–25 Methods You Typically Write on ViewRowImpl

If you want to...	Write a method like this in your custom ViewRowImpl class
Calculate the value of a view object level transient attribute	<pre>getAttrName()</pre> <p>Note: JDeveloper generates the skeleton of the method for you, but you need to write the custom calculation logic inside the method body.</p>
Perform custom processing of the setting of a view row attribute	<pre>setAttrName()</pre> <p>Note: JDeveloper generates the skeleton of the method for you, but you need to write the custom logic inside the method body if required.</p>
Determine the updateability of an attribute in a conditional way.	<pre>isAttributeUpdateable()</pre>
Custom methods that expose logical operations on the current row, optionally callable by clients	<pre>doSomething()</pre> <p>Note: Often these view-row level custom methods simply turn around and delegate to a method call on the underlying entity object related to the current row.</p>

JDeveloper can generate you a custom *YourViewObjectNameRow* interface containing view row custom methods that you've chosen to expose to the client. You can accomplish this by visiting the **Client Row Interface** tab of the View Object editor, and shuttling the methods you'd like to appear in your client interface into the **Selected** list.

E.2.7.3 Methods You Typically Override in Your Custom ViewRowImpl Subclass

Table E–26 Methods You Typically Override in Your Custom ViewRowImpl Subclass

If you want to...	Write a method like this in your custom ViewRowImpl class
Determine the updateability of an attribute in a conditional way.	<pre>isAttributeUpdateable()</pre>

E.2.8 Setting Up Your Own Layer of Framework Base Classes

Before you begin to develop application specific business components, Oracle recommends creating yourself a layer of classes that extend all of the ADF Business Components framework base implementation classes described in this paper. An example of a customized framework base class for application module components might look like this:

```
package com.yourcompany.adfextensions;
import oracle.jbo.server.ApplicationModuleImpl;
public class CustomApplicationModuleImpl extends ApplicationModuleImpl {
    /*
     * We might not yet have any custom code to put here yet, but
     * the first time we need to add a generic feature that all of
     * our company's application modules need, we will be very happy
     * that we thought ahead to leave ourselves a convenient place
     * in our class hierarchy to add it so that all of the application
     * modules we have created will instantly benefit by that new feature,
     * behavior change, or even perhaps, bug workaround.
    */
}
```

A common set of customized framework base classes in a package name of your own choosing like *com.yourcompany.adfextensions*, each importing the *oracle.jbo.server.** package, would consist of the following classes.

- public class CustomEntityImpl extends EntityImpl
- public class CustomEntityDefImpl extends EntityDefImpl
- public class CustomViewObjectImpl extends ViewObjectImpl
- public class CustomViewRowImpl extends ViewRowImpl
- public class CustomApplicationModuleImpl extends ApplicationModuleImpl
- public class CustomDBTransactionImpl extends DBTransactionImpl2
- public class CustomDatabaseTransactionFactory extends DatabaseTransactionFactory

For completeness, you may also want to create customized framework classes for the following classes as well, but overriding anything in these classes would be a fairly rare requirement.

- public class CustomViewDefImpl extends ViewDefImpl
- public class CustomEntityCache extends EntityCache
- public class CustomApplicationModuleDefImpl extends ApplicationModuleDefImpl

ADF Business Components Java EE Design Pattern Catalog

This appendix provides a brief summary of the Java Platform, Enterprise Edition (Java EE) design patterns that the ADF Business Components layer implements for you.

This appendix contains the following section:

- [Section F, "ADF Business Components Java EE Design Pattern Catalog"](#)

F.1 Java EE Design Patterns Implemented by ADF Business Components

By using the Oracle Application Development Framework's business components building-blocks and related design-time extensions to JDeveloper, you get a prescriptive architecture for building richly-functional and cleanly layered Java EE business services with great performance. [Table F-1](#) provides a brief overview of the numerous design patterns that the ADF Business Components layer implements for you. Some are the familiar patterns from Sun's Java EE BluePrints and some are design patterns that ADF Business Components adds to the list. For details about Java EE BluePrints, see the BluePrints page at the Sun Developer Network website at <http://java.sun.com/reference/blueprints/>.

Table F-1 Java EE Design Patterns Implemented by ADF Business Components

Pattern Name and Description	How ADF Business Components Implements It
Model/View/Controller Cleanly separates the roles of data and presentation, allowing multiple types of client displays to work with the same business information.	The ADF Application Module provides a generic implementation of a Model/View/Controller "application object" that simplifies exposing the application data model for any application or service, and facilitates declaratively specifying the boundaries of a logical unit of work. Additional UI-centric frameworks and tag libraries provided in JDeveloper 10g help the developer implement the View and Controller layers.
Interface / Implementation Separation Cleanly separates the API or Interface for components from their implementation class.	ADF Business Components enforce a logical separation of client-tier accessible functionality (via interfaces) and their business tier implementation. JDeveloper handles the creation of custom interfaces and client proxy classes automatically.

Table F-1 (Cont.) Java EE Design Patterns Implemented by ADF Business Components

Pattern Name and Description	How ADF Business Components Implements It
Service Locator Abstracts the technical details of locating a service so the client and use it more easily.	ADF application modules are looked up using a simple configuration object which hides the low-level details of finding the service instance behind the scenes. For Fusion web applications, it also hides the implementation of the application module pool usage, a lightweight pool of service components that improves application scalability.
Inversion of Control A containing component orchestrates the lifecycle of the components it contains, invoking specific methods that the developer can overrides at the appropriate times so the developer can focus more on what the code should do instead when it should get executed.	ADF components contain a number of easy-to-override methods that the framework invokes as needed during the course of application processing.
Dependency Injection Simplifies application code, and increases configuration flexibility by deferring component configuration and assembly to the container.	The ADF configures all its components from externalized XML metadata definition files. The framework automatically injects dependent objects like view object instances into your application module service component and entity objects into your view rows at runtime, implementing lazy loading. It supports runtime factory substitution of components by any customized subclass of that component to simplify on-site application customization scenarios. Much of the ADF functionality is implemented via dynamic injection of validator and listener subscriptions that coordinate the framework interactions depending on what declarative features have been configured for each component in their XML metadata.
Active Record Avoids the complexity of "anything to anything" object/relational mapping, by providing an object that wraps a row in a database table or view, encapsulates the database access, and adds domain logic on that data.	ADF entity objects handle the database mapping functionality you use most frequently, including inheritance, association, and composition support, without having to think about object/relational mapping. They also provide a place to encapsulate both declarative business rules and one-off programmatic business domain logic.
Data Access Objects Avoids unnecessary marshalling overhead by implementing dependent objects as lightweight, persistent classes instead of each as an Enterprise Bean. Isolates persistence details into a single, easy to maintain class.	ADF view objects automate the implementation of data access for reading data using SQL statements. ADF entity objects automate persistent storage of lightweight business entities. ADF view objects and entity objects cooperate to provide a sophisticated, performant data access objects layer where any data queried through a view object can optionally be made fully updatable without writing any "application plumbing" code.
Session Facade Avoids inefficient client access of Entity Beans and inadvertent exposure of sensitive business information by wrapping Entity Beans with a Session Bean.	ADF application modules are designed to implement a coarse-grained "service facade" architecture in any of their supported deployment modes. When deployed as EJB Session Beans or as a Service Interface, they provide an implementation of the Session Facade pattern automatically.

Table F-1 (Cont.) Java EE Design Patterns Implemented by ADF Business Components

Pattern Name and Description	How ADF Business Components Implements It
Value Object	ADF provides an implementation of a generic Row object, which is a metadata-driven container of any number and kind of attributes that need to be accessed by a client. The developer can work with the generic Row interface and do late-bound <code>getAttribute("Price")</code> and <code>setAttribute("Quantity")</code> calls, or optionally generate early-bound row interfaces like <code>OverdueOrdersRow</code> , to enable type-safe method calls like <code>getPrice()</code> and <code>setQuantity()</code> . Smarter than just a simple "bag 'o attributes" the ADF Row object can be introspected at runtime to describe the number, names, and types of the attributes in the row, enabling sophisticated, generic solutions to be implemented.
Page-by-Page Iterator	ADF provides an implementation of a generic RowSet interface which manages result sets produced by executing View Object SQL queries. RowSet allows the developer to set a desired page-size, for example 10 rows, and page up and down through the query results in these page-sized chunks. Since data is retrieved lazily, only data the user actually visits will ever be retrieved from the database on the backend, and in the client tier the number of rows in the page can be returned over the network in a single roundtrip.
Fast-Lane Reader	ADF View Objects read data directly from the database for best performance, however they give developers a choice regarding data consistency. If updateability and/or consistency with pending changes is desired, the developer need only associate his/hew View Object with the appropriate Entity Objects whose business data is being presented. If consistency is not a concern, View Objects can simply perform the query with no additional overhead. In either case, the developer never has to write JDBC data access code. They only provide appropriate SQL statements in XML descriptors.
(Bean) Factory	All ADF component instantiation is done based on XML configuration metadata through factory classes allowing runtime substitution of specialized components to facilitate application customization.
Entity Facade	The ADF view object can surface any set of attributes and methods from any combination of one or more underlying entity objects to furnish the client with a single, logical value object to work with.

Table F-1 (Cont.) Java EE Design Patterns Implemented by ADF Business Components

Pattern Name and Description	How ADF Business Components Implements It
Value Messenger	ADF's value object implementation coordinates with a client-side value object cache to batch attribute changes to the EJB tier and receive batch attribute updates which occur as a result of middle-tier business logic. The ADF Value Messenger implementation is designed to not require any kind of asynchronous messaging to achieve this effect.
Continuations	ADF's application module pooling and state management functionality combine to deliver this developer value-add. This avoids dedicating application server tier resources to individual users, but supports a "stateless with user affinity" optimization that you can tune.

Performing Common Oracle Forms Tasks in Oracle ADF

This appendix describes how some common Oracle Forms tasks are implemented in Oracle ADF. In Oracle Forms, you do some tasks in the data block, and others in the UI. For this reason, the appendix is divided into two sections: tasks that relate to data, and tasks that relate to the UI.

This appendix contains the following sections:

- [Section G.1, "Performing Tasks Related to Data"](#)
- [Section G.2, "Performing Tasks Related to the User Interface"](#)

G.1 Performing Tasks Related to Data

In Oracle Forms, tasks that relate solely to data are performed in the data block. In Oracle ADF, these tasks are done on the business components that persist data (entity objects) and the objects that query data (view objects).

G.1.1 How to Retrieve Lookup Display Values for Foreign Keys

In Oracle Forms, an editable table often has foreign key lookup columns to other tables. The user-friendly display values corresponding to the foreign key column values exist in related tables. You often need to present these related display values to the user.

In Oracle Forms, this was a complicated task that required adding nondatabase items to the data block, adding a block-level POST-QUERY trigger to the data block, and writing a SQL select statement for each foreign key attribute. Additionally, if the user changed the data, you needed to sync the foreign key values with an item-level WHEN-VALIDATE-ITEM trigger. This process is much easier in Oracle ADF.

Implementation of the task in Oracle ADF

1. Create a view object that includes the main, editable entity object as the primary entity usage, and includes secondary "reference" entity usages for the one or more associated entities whose underlying tables contain the display text. For more information, see [Section 5.5.1, "How to Create Joins for Entity-Based View Objects"](#).
2. Select the desired attributes (at least the display text) from the secondary entity usages as described in [Section 5.5.2, "How to Select Additional Attributes from Reference Entity Usages"](#).

At runtime, the data for the main entity and all related lookup display fields are retrieved from the database in a single join.

If the user can change the data, no additional steps are required. If the user changes the value of a foreign key attribute, the reference information is automatically retrieved for the new, related row in the associated table.

G.1.2 How to Get the Sysdate from the Database

In Oracle Forms, when you wanted to get the current date and time, you got the `sysdate` from the database. In Oracle ADF, you also have the option of getting the system date using a Java method or a Groovy expression.

Implementation of the task in Oracle ADF

To get the system date from the database, you can use the following Groovy expression at the entity level:

```
DBTransaction.currentDbTime
```

Note: The `DBTransaction` reference is for entity-level Groovy expressions only.

If you want to assign a default value to an attribute using this Groovy expression, see [Section 4.10.7, "How to Define a Default Value Using a Groovy Expression"](#).

To get the system date from Java, you call the `getCurrentDate()` method. For more information, see [Section 8.10, "Accessing the Current Date and Time"](#).

G.1.3 How to Implement an Isolation Mode That Is Not Read Consistent

In Oracle Forms, you might have been concerned with read consistency, that is, the ability of the database to deliver the state of the data at the time the SQL statement was issued.

Implementation of the task in Oracle ADF

If you use an entity-based view object, the query sees the changes currently in progress by the current user's session in the pending transaction. This is the default behavior, and the most accurate.

If instead, you want a snapshot of the data on the database without considering the pending changes made by the current user, you can use a read-only view object and reexecute the query to see the latest committed database values. For more information on read-only view objects, see [Section 5.2.3, "How to Create an Expert Mode, Read-Only View Object"](#).

G.1.4 How to Implement Calculated Fields

Calculated fields are often used to show the sum of two values, but could also be used for the concatenated value of two or more fields, or the result of a method call.

Implementation of the task in Oracle ADF

Calculated attributes are usually not stored in the database, as their values can easily be obtained programmatically. Attributes that are used in the middle tier, but that are

not stored in the database are called *transient* attributes. Transient attributes can be defined at the entity object level or the view object level.

If a transient attribute will be used by more than one view object that might be based on an entity object, then define the attribute at the entity object level. Otherwise, define the transient attribute at the view object level for a particular view object.

To define transient attributes at the entity object level, see [Section 4.13, "Adding Transient and Calculated Attributes to an Entity Object"](#). To define transient attributes at the view object level, see [Section 5.13, "Adding Calculated and Transient Attributes to a View Object"](#).

G.1.5 How to Implement Mirrored Items

In Oracle Forms, you may be used to using mirrored items to show two or more fields that share identical values.

Implementation of the task in Oracle ADF

There is no need to have mirrored items in Oracle ADF, because the UI and data are separated. The same view object can appear on any number of pages, so you don't need to create mirrored items that have the same value. Likewise, a form could have the same field represented in more than one place and it would not have to be mirrored.

G.1.6 How to Use Database Columns of Type CLOB or BLOB

If you are used to working with standard database types, you may be wondering how to use the CLOB and BLOB types in Oracle ADF.

Implementation of the task in Oracle ADF

In Oracle ADF, use the built-in data types `ClobDomain` or `BlobDomain`. These are automatically created when you reverse-engineer entity objects or view objects from existing tables with these column types. ADF Business Components also supports data types for Intermedia column types: `OrdImage`, `OrdAudio`, `OrdDoc`, and `OrdVideo`. For more information, see [Section 4.10.1, "How to Set Database and Java Data Types for an Entity Object Attribute"](#).

G.2 Performing Tasks Related to the User Interface

This section describes how to perform some common Oracle Forms tasks that relate to the UI with Oracle ADF.

G.2.1 How to Lay Out a Page

Oracle Forms is based on an absolute pixel or point-based layout, as compared to the container-based approach of JSF, and the Layout Manager approach in ADF Swing.

Implementation of the task in Oracle ADF

See the *Oracle Fusion Middleware Web User Interface Developer's Guide for Oracle Application Development Framework* for information on how to lay out a page in Oracle ADF.

G.2.2 How to Stack Canvases

In Oracle Forms, stacked canvases are often used to hide and display areas of the screen.

Implementation of the task in Oracle ADF

The analog of stacked canvases in Oracle ADF is panels (layout containers) with the rendered property set to true or false. See the *Oracle Fusion Middleware Web User Interface Developer's Guide for Oracle Application Development Framework* for more information.

G.2.3 How to Implement a Master-Detail Screen

Master-detail relationships in Oracle ADF are coordinated through a view link. A view link is conceptually similar to a Oracle Forms *relation*.

Implementation of the task in Oracle ADF

For information on how to create view links, see [Section 5.6, "Working with Multiple Tables in a Master-Detail Hierarchy"](#). Once you have established a relationship between two view objects with a view link, see [Section 5.6.4, "How to Enable Active Master-Detail Coordination in the Data Model"](#).

G.2.4 How to Implement an Enter Query Screen

In Oracle Forms, another common task is creating an *enter query screen*. That is, a screen that starts in Find mode.

Implementation of the task in Oracle ADF

Complete information on how to create a search form is covered in [Chapter 25, "Creating ADF Databound Search Forms"](#). In particular, you may want to look at [Section 25.3.1, "How to Set Search Form Properties on the View Criteria"](#).

G.2.5 How to Implement an Updatable Multi-Record Table

In Oracle Forms, you may be used to creating tables where you can edit and insert many records at the same time. In Oracle ADF, this is slightly more complicated when using a JSF page, because the operations to edit an existing record and to create a new record are not the same.

Implementation of the task in Oracle ADF

In Oracle ADF, this is done using an input table. To create an input table, see [Section 21.4, "Creating an Input Table"](#).

G.2.6 How to Create a Popup List of Values

In Oracle Forms, it is simple to create a list of values (LOV) object and then associate that object with a field in a declarative manner. This LOV would display a popup window and provide the following capabilities:

- Selection of modal values
- Query area at the top of the LOV dialog
- Display of multiple columns
- Automatic reduction of LOV contents, possibly based on the contents of the field that launched the LOV

- Automatic selection of the list value when only one value matches the value in the field when the LOV function is invoked
- Validation of the field value based on the values cached by the LOV
- Automatic popup of the LOV if the field contents are not valid

Implementation of the task in Oracle ADF

To implement a popup list in Oracle ADF, you configure one of the view object's attributes to be of LOV type, and select **Input Text with List of Values** as the style for its UI hint. For a description of how to do this, see [Section 5.11, "Working with List of Values \(LOV\) in View Object Attributes"](#).

G.2.7 How to Implement a Dropdown List as a List of Values

In Oracle Forms, you can create a list of values (LOV) object and then associate that object with a field in a declarative manner. In Oracle ADF, you can implement an LOV (look-up-value) screen with a search item, usable for a look-up field with many possible values.

Implementation of the task in Oracle ADF

To implement a dropdown list in Oracle ADF, you configure one of the view object's attributes to be of LOV type, and select **Input Text with List of Values** as the style for its UI hint. For a description of how to do this, see [Section 5.11, "Working with List of Values \(LOV\) in View Object Attributes"](#).

G.2.8 How to Implement a Dropdown List with Values from Another Table

In Oracle Forms, you can create a list of values (LOV) object and then associate that object with a field in a declarative manner. In Oracle ADF, you can implement a dropdown list with string values from a different table that populates the field with an id code that is valid input in the table that the screen is based on.

Implementation of the task in Oracle ADF

To implement a dropdown list of this type in Oracle ADF, you configure one of the view object's attributes to be of LOV type, and select **Choice List** as the style for its UI hint. For a description of how to do this, see [Section 5.11, "Working with List of Values \(LOV\) in View Object Attributes"](#).

G.2.9 How to Implement Immediate Locking

In Oracle ADF, you can lock a record in the database at the first moment it is obvious that the user is going to change a specific record.

Implementation of the task in Oracle ADF

Immediate locking mode is the default in ADF Business Components, although it is not typically used in web application scenarios. For web applications, use `jbo.locking.mode=optimistic`. For more information, see [Section 36.11.1, "How to Set Applications to Use Optimistic Locking"](#).

G.2.10 How to Throw an Error When a Record Is Locked

When a record has been locked by a user, it's helpful to throw an error to let other users know that the record is not currently updatable.

Implementation of the task in Oracle ADF

Locking rows and throwing an exception if the row is already locked is built-in ADF Business Components functionality. There are a couple of different ways that you can handle the error message, depending on whether you want a static error message or a custom message with information about the current row.

- To throw a static message, register a custom message bundle in your ADF Business Components project to substitute the default `RowAlreadyLockedException`'s error message, to something more meaningful or user-friendly.
- To throw a message that contains information about the row, override the `lock()` method on the entity object, using a try/catch block to catch the `RowAlreadyLocked` exception. After you catch the exception, you can throw an error message that might contain more specific information about the current row.