



Quick answers to common problems

Oracle JDeveloper 11gR2 Cookbook

Over 85 simple but incredibly effective recipes for using Oracle JDeveloper 11gR2 to build ADF applications

Foreword by Frank Nimphius

Senior Principal Product Manager, Oracle Application Development Tools

Nick Haralabidis

[PACKT] enterprise 
PUBLISHING professional expertise distilled

Oracle JDeveloper 11gR2 Cookbook

Over 85 simple but incredibly effective recipes for using
Oracle JDeveloper 11gR2 to build ADF applications

Nick Haralabidis

[PACKT] enterprise 
PUBLISHING professional expertise distilled

BIRMINGHAM - MUMBAI

Oracle JDeveloper 11gR2 Cookbook

Copyright © 2012 Packt Publishing

All rights reserved. No part of this book may be reproduced, stored in a retrieval system, or transmitted in any form or by any means, without the prior written permission of the publisher, except in the case of brief quotations embedded in critical articles or reviews.

Every effort has been made in the preparation of this book to ensure the accuracy of the information presented. However, the information contained in this book is sold without warranty, either express or implied. Neither the author, nor Packt Publishing, and its dealers and distributors will be held liable for any damages caused or alleged to be caused directly or indirectly by this book.

Packt Publishing has endeavored to provide trademark information about all of the companies and products mentioned in this book by the appropriate use of capitals. However, Packt Publishing cannot guarantee the accuracy of this information.

First published: January 2012

Production Reference: 1170112

Published by Packt Publishing Ltd.
Livery Place
35 Livery Street
Birmingham B3 2PB, UK.

ISBN 978-1-84968-476-7

www.packtpub.com

Cover Image by Artie Ng (artherng@yahoo.com.au)

Credits

Author

Nick Haralabidis

Reviewers

Edwin Biemond

Spyros Doulgeridis

Frank Nimphius

Acquisition Editor

Stephanie Moss

Lead Technical Editor

Meeta Rajani

Technical Editors

Sonali Tharwani

Vishal D'souza

Copy Editor

Laxmi Subramanian

Project Coordinator

Leena Purkait

Proofreader

Dan McMahon

Indexers

Hemangini Bari

Monica Ajmera Mehta

Tejal Daruwale

Production Coordinator

Arvindkumar Gupta

Cover Work

Arvindkumar Gupta

Foreword

Oracle has a long and successful history of building enterprise application development tools, including some that have outlived their competition. As a former Oracle Forms Product Manager and current Oracle JDeveloper and Oracle Application Development Framework (ADF) Product Manager, this part of the Oracle history has been mine for the last 15 years, and I'm very grateful that there's currently no end in sight!

Building enterprise applications based on Java EE standards is a well-accepted and understood concept for building Rich Internet Applications (RIA) and service-oriented user interfaces. While Java language skills are standard knowledge for college graduates, broader topics, such as service-enablement, persistence, application security, customization, portalization, and so on are not always so well understood. Adding to this, the "framework-of-the-day" problem—in which framework solutions quickly wax and wane in popularity—makes it difficult for enterprises to adopt software. What most enterprise businesses require is the benefit of standards, but with an end-to-end framework that provides a stable and consistent interface, which can abstract away future technology shifts.

Proven by Oracle Fusion Applications and customer success, Oracle ADF fulfills that need: a rapid application development environment that reduces the skills required for building modern rich enterprise applications to a single learning curve.

Technically, Oracle ADF is an end-to-end Java EE framework for building rich enterprise web and mobile applications based on Java EE services and SOA. Oracle ADF integrates existing Java frameworks into a single architecture and a fully integrated declarative development environment that shields developers from low-level API programming.

Besides being used by Oracle Fusion Applications and Oracle customers, Oracle ADF is at the heart of Oracle Middleware and is the technology of choice for building Fusion Middleware (FMW) products, such as Enterprise Manager, WebCenter, UCM, BPM, BI, and so on, showing Oracle's commitment to ADF.

Technology alone, however, is no guarantee for success. Community acceptance and contribution is also an important backbone and measurement of software frameworks and products, including Oracle ADF.

Oracle ADF is supported by a very active and growing community of bloggers, forum posters, and speakers, as well as book and article authors. The *Oracle JDeveloper 11gR2 Cookbook* you hold in your hands is another example of the ongoing contribution from the ADF community by author Nick Haralabidis.

The book is a practical guide to learning Oracle ADF, providing code solutions, and technical explanations to common Oracle ADF questions and developer challenges. Being one of the technical reviewers for this book and having written other titles as an author myself, I appreciate the time, effort, and dedication Nick Haralabidis has put into writing this book, as well as the Oracle ADF expertise and practices he shares with you, the reader. This book is not a beginner's guide, but a useful reference for all developers starting enterprise application development with Oracle ADF.

Frank Nimphius

Senior Principal Product Manager, Oracle Application Development Tools

About the Author

Nick Haralabidis has over 20 years experience in the Information Technology industry and a multifaceted career in positions such as Senior IT Consultant, Senior Software Engineer, and Project Manager for a number of U.S. and Greek corporations (Compuware, Chemical Abstracts Service, NewsBank, CheckFree, Intrasoft International, Unisystems, MedNet International, and others). His many years of experience have exposed him to a wide range of technologies, such as Java, J2EE, C++, C, Tuxedo, and a number of other database technologies.

For the last four years, Nick is actively involved in large implementations of next generation enterprise applications utilizing Oracle's JDeveloper, Application Development Framework (ADF), and SOA technologies.

He holds a B.S. in Computer Engineering and a M.S. in Computer Science from the University of Bridgeport.

When he is not pursuing ADF professionally, he writes on his blogs *JDeveloper Frequently Asked Questions* (<http://jdeveloperfaq.blogspot.com>) and *ADF Code Bits* (<http://adfcodebits.blogspot.com>). He is active at the *Oracle Technology Network (OTN) JDeveloper and ADF* forum where he both learns and helps.

To Aphrodite, Konstantina and Margaritta, my true inspirations.

To the Packt team and especially to Stephanie Moss for her trust, encouragement, and direction.

To the book reviewers, Frank Nimphius, Edwin Biemond, and Spyros Doulgeridis for their time, expertise, and invaluable insight.

About the Reviewers

Edwin Biemond is an Oracle ACE and Solution Architect at Amis, specializing in messaging with Oracle SOA Suite and Oracle Service Bus. He is an expert in ADF development, WebLogic Administration, high availability, and security. His Oracle career began in 1997, where he was developing an ERP, CRM system with Oracle tools. Since 2001, Edwin has changed his focus to integration, security, and Java development. Edwin was awarded with Java Developer of the year 2009 by Oracle Magazine, won the EMEA Oracle Partner Community Award in 2010, and contributed some content to the Oracle SOA Handbook of Lucas Jellema. He is an international speaker at Oracle OpenWorld & ODTUG and has a popular blog called Java/Oracle SOA blog at <http://biemond.blogspot.com>.

Spyros Doulgeridis holds two M.Sc. degrees, one in Telecommunication from Brunel University in the U.K. and one in Software Engineering from N.T.U.A. in Greece. With proven experience using major Java frameworks in JEE applications, he has been working with Oracle technologies, and especially ADF 11g, since 2008 in a major Form to ADF migration project—one of Oracle's Success Stories. During this project, he had many roles including ADF developer, designer of Forms to ADF migration, ADF/Java reviewer, and was responsible for the application's build process and deployment on Weblogic Server. He likes to share his experiences by blogging on adfhowto.blogspot.com.

I would like to thank Packt Publishing and especially Mrs. Stephanie Moss for giving me the opportunity to work on this book. Also, I would like to thank the author for this interesting journey into Oracle ADF through his helpful and practical recipes. Finally and above all, I would like to thank all of those close to me, who missed me while working on this book.

Frank Nimphius is a Senior Principal Product Manager in the Oracle Application Development Tools group at Oracle Corporation, where he specializes in Oracle JDeveloper and the Oracle Application Development Framework (ADF).

As a speaker, Frank represents the Oracle ADF and Oracle JDeveloper development team at user conferences world-wide. Frank owns the ADF Code Corner website (<http://www.oracle.com/technetwork/developer-tools/adf/learnmore/index-101235.html>), and the "OTN Forum Harvest" blog (<http://blogs.oracle.com/jdevotnharvest/>).

As an author, Frank frequently writes for Oracle Magazine and co-authored the "Oracle Fusion Developer Guide" book published in 2009 by McGraw Hill.

www.PacktPub.com

Support files, eBooks, discount offers and more

You might want to visit www.PacktPub.com for support files and downloads related to your book.

Did you know that Packt offers eBook versions of every book published, with PDF and ePub files available? You can upgrade to the eBook version at www.PacktPub.com and as a print book customer, you are entitled to a discount on the eBook copy. Get in touch with us at service@packtpub.com for more details.

At www.PacktPub.com, you can also read a collection of free technical articles, sign up for a range of free newsletters and receive exclusive discounts and offers on Packt books and eBooks.



<http://PacktLib.PacktPub.com>

Do you need instant solutions to your IT questions? PacktLib is Packt's online digital book library. Here, you can access, read and search across Packt's entire library of books.

Why Subscribe?

- ▶ Fully searchable across every book published by Packt
- ▶ Copy and paste, print and bookmark content
- ▶ On demand and accessible via web browser

Free Access for Packt account holders

If you have an account with Packt at www.PacktPub.com, you can use this to access PacktLib today and view nine entirely free books. Simply use your login credentials for immediate access.

Instant Updates on New Packt Books

Get notified! Find out when new books are published by following [@PacktEnterprise](https://twitter.com/PacktEnterprise) on Twitter, or the *Packt Enterprise* Facebook page.

Table of Contents

Preface	1
Chapter 1: Prerequisites to Success: ADF Project Setup and Foundations	7
Introduction	8
Installation of JDeveloper on Linux	8
Breaking up the application in multiple workspaces	12
Setting up BC base classes	18
Setting up logging	22
Using a custom exception class	27
Using ADFUtils/JSFUtils	32
Using page templates	35
Using a generic backing bean actions framework	42
Chapter 2: Dealing with Basics: Entity Objects	47
Introduction	47
Using a custom property to populate a sequence attribute	48
Overriding doDML() to populate an attribute with a gapless sequence	51
Creating and applying property sets	54
Using getPostedAttribute() to determine the posted attribute's value	58
Overriding remove() to delete associated children entities	60
Overriding remove() to delete a parent entity in an association	63
Using a method validator based on a view object accessor	66
Using Groovy expressions to resolve validation error message tokens	70
Using doDML() to enforce a detail record for a new master record	73
Chapter 3: A Different Point of View: View Object Techniques	75
Introduction	75
Iterating a view object using a secondary rowset iterator	76
Setting default values for view row attributes	81
Controlling the updatability of view object attributes programmatically	84

Setting the Queryable property of a view object attribute programmatically	86
Using a transient attribute to indicate a new view object row	88
Conditionally inserting new rows at the end of the rowset	90
Using findAndSetCurrentRowByKey() to set the view object currency	92
Restoring the current row after a transaction rollback	95
Dynamically changing the WHERE clause of the view object query	99
Removing a row from a rowset without deleting it from the database	101
Chapter 4: Important Contributors: List of Values, Bind Variables, View Criteria	105
Introduction	106
Setting up multiple LOVs using a switcher attribute	106
Setting up cascading LOVs	110
Creating static LOVs	116
Overriding bindParametersForCollection() to set a view object bind variable	118
Creating view criteria programmatically	122
Clearing the values of bind variables associated with the view criteria	126
Searching case insensitively using view criteria	128
Chapter 5: Putting them all together: Application Modules	131
Introduction	131
Creating and using generic extension interfaces	132
Exposing a custom method as a web service	135
Accessing a service interface method from another application module	139
A passivation/activation framework for custom session-specific data	143
Displaying application module pool statistics	151
Using a shared application module for static lookup data	156
Using a custom database transaction	159
Chapter 6: Go with the Flow: Task Flows	163
Introduction	163
Using an application module function to initialize a page	164
Using a task flow initializer to initialize a task flow	170
Calling a task flow as a URL programmatically	176
Retrieving the task flow definition programmatically using MetadataService	182
Creating a train	186
Chapter 7: Face Value: ADF Faces, JSF Pages, and User Interface Components	193
Introduction	194
Using an af:query component to construct a search page	194

Using an af:pop-up component to edit a table row	198
Using an af:tree component	205
Using an af:selectManyShuttle component	210
Using an af:carousel component	215
Using an af:poll component to periodically refresh a table	219
Using page templates for pop-up reuse	222
Exporting data to a client file	228
Chapter 8: Baking not Baking: Bean Recipes	233
Introduction	234
Determining whether the current transaction has pending changes	234
Using a custom af:table selection listener	236
Using a custom af:query listener to allow execution of a custom application module operation	239
Using a custom af:query operation listener to clear both the query criteria and results	243
Using a session scope bean to preserve session-wide information	248
Using an af:popup during long running tasks	252
Using an af:popup to handle pending changes	255
Using an af:iterator to add pagination support to a collection	259
Chapter 9: Handling Security, Session Timeouts, Exceptions, and Errors	265
Introduction	266
Enabling ADF security	266
Using a custom login page	272
Accessing the application's security information	275
Using OPSS to retrieve the authenticated user's profile from the identity store	279
Detecting and handling session timeouts	285
Using a custom error handler to customize how exceptions are reported to the ViewController	288
Customizing the error message details	291
Overriding attribute validation exceptions	295
Chapter 10: Deploying ADF Applications	299
Introduction	299
Configuring and using the Standalone WebLogic Server	300
Deploying on the Standalone WebLogic Server	306
Using ojdeploy to automate the build process	311
Using Hudson as a continuous integration framework	316

Chapter 11: Refactoring, Debugging, Profiling, and Testing	323
Introduction	323
Synchronizing business components with database changes	324
Refactoring ADF components	327
Configuring and using remote debugging	329
Logging Groovy expressions	333
Dynamically configuring logging in WebLogic Server	335
Performing log analysis	337
Using CPU profiler for an application running on a standalone WebLogic server	339
Configuring and using JUnit for unit testing	343
Chapter 12: Optimizing, Fine-tuning, and Monitoring	347
Introduction	347
Using Update Batching for entity objects	348
Limiting the rows fetched by a view object	350
Limiting large view object query result sets	352
Limiting large view object query result sets by using required view criteria	356
Using a work manager for processing of long running tasks	358
Monitoring the application using JRockit Mission Control	369
Chapter 13: Miscellaneous Recipes	
This chapter is not present in the book but is available as a free download from: http://www.packtpub.com/sites/default/files/downloads/4767EN_Chapter_13_Miscellaneous_Recipes.pdf	
Index	373

Preface

This book contains a wealth of resources covering Oracle's JDeveloper 11g release and the Application Development Framework (ADF) and how these technologies can be used for the design, construction, testing, and optimizing of Fusion web applications. Being vast and complex technologies, an attempt has been made to cover a wide range of topics related specifically to Fusion web applications development with ADF, utilizing the complete ADF stack. These topics are presented in the form of recipes, many of them derived from the author's working experience covering real world use cases. The topics include, but are not limited to, foundational recipes related to laying out the project groundwork, recipes related to the ADF business components, recipes related to ViewController, recipes related to security, optimization and so on.

In the maze of information related to Fusion web applications development with ADF, it is the author's hope that aspiring ADF developers will find in this book some of the information they are looking for. So lift up your sleeves, put on your ADF chef's hat, pick up a recipe or two, and let's start cooking!

What this book covers

Chapter 1, Pre-requisites to Success: ADF Project Setup and Foundations, covers a number of recipes related to foundational concepts of Fusion web application development with ADF. By applying and expanding these recipes during the early architectural and design phases as needed, subsequent application development takes on a form, a structure, and the necessary uniformity. Many if not most of the recipes in the following chapters rely on these recipes.

Chapter 2, Dealing with Basics: Entity Objects, starts our journey into the world of ADF business components. First stop: entity objects. The recipes in this chapter deal with some of the most common framework functionality that is overridden in real world applications to provide customized business functionality.

Chapter 3, A Different Point of View: View Objects Techniques, covers a number of recipes related to view objects. This chapter explains how to control attribute updatability, how to set attribute default values, how to iterate view object row sets, and many more.

Chapter 4, Important Contributors: List of Values, Bind Variables, View Criteria, covers additional topics related to view objects. These topics include recipes related to list of values (LOVs), bind variables and view criteria. The reader will learn, among other things, how to setup multiple LOVs using a switcher attribute, cascading and static LOVs, and how to create view criteria programmatically.

Chapter 5, Putting them all together: Application Modules, includes a number of recipes related to application modules. You will learn, among others, how to create and use generic extension interfaces, expose a custom application module method as a web service and access a service interface from another application module. Additional recipes cover topics such as a passivation/activation framework, using shared application modules for static lookup data and custom database transactions.

Chapter 6, Go with the flow: Task Flows, delves into the world of ADF task flows. Among others, you will learn how to use an application module function as a method call to initialize a page, how to use a task flow initializer, how to retrieve the task flow definition programmatically and how to create a train.

Chapter 7, Face Value: ADF Faces, JSPX Pages and Components, includes recipes detailing the use of a variety of ADF Faces components, such as the query component, the popup window component, the tree component, the select many shuttle component, the carousel component, and others.

Chapter 8, Backing not Baking: Bean Recipes, introduces topics related to backing beans. A number of topics are covered including the use of custom table selection listeners, custom query and query operation listeners, session beans to preserve session-wide information, popup windows to handle long running tasks.

Chapter 9, Handling Security, Session Timeouts, Exceptions and Errors, covers topics related to handling security, session timeouts, exceptions and errors for an ADF Fusion web application. The recipes in this chapter will show the reader how to enable ADF security, how to use a custom login page, how to access the application's security information, how to detect and handle session timeouts, and how to use a custom error handler.

Chapter 10, Deploying ADF Applications, includes recipes related to the deployment of ADF Fusion web applications. These recipes include the configuration and use of the standalone WebLogic server, the deployment of applications on the standalone WebLogic server, the use of the ojdeploy tool and the use of Hudson as a continuous integration framework.

Chapter 11, Refactoring, Debugging, Profiling, Testing, deals with topics related to refactoring, debugging, profiling, and testing ADF Fusion web applications. The recipes in this chapter cover topics such as the synchronization of business components to changes in the database, refactoring of ADF components, configuring and using remote debugging, configuring logging in the WebLogic server, CPU profiling and the configuration, and usage of JUnit for unit testing.

Chapter 12, Optimizing, Fine-tuning and Monitoring, covers topics related to optimizing, fine-tuning, and monitoring ADF Fusion web applications. The recipes in this chapter demonstrate how to limit the rows fetched by view objects, how to limit large view object queries, how to use work managers for processing long-running tasks and how to monitor your application using the JRockit Mission Control.

Chapter 13, Miscellaneous Recipes, the additional content recipes cover topics related among others to using JasperReports, uploading images to the server, and handling and customizing database-related errors. This chapter is not present in the book but is available as a free download from the following link: http://www.packtpub.com/sites/default/files/downloads/4767EN_Chapter_13_Miscellaneous_Recipes.pdf.

What you need for this book

The recipes in this book utilize the latest release of JDeveloper at the time of writing, namely JDeveloper 11g R2 11.1.2.1.0. This release of JDeveloper comes bundled with the necessary ADF libraries and a standalone installation of the WebLogic server. Ensure that the WebLogic server is installed as part of the JDeveloper installation.

In addition, you will need a database connection to Oracle's HR schema. This schema is provided along with the Oracle XE database.

A number of recipes cover topics that will require you to download and install the following additional software: Hudson continuous integration, JRockit Mission Control, Jasper Reports, and iReport.

Who this book is for

This book is targeted to intermediate or advanced developers, designers and architects already utilizing JDeveloper, the ADF framework, and Oracle's Fusion technologies. Developers utilizing the complete ADF stack for building ADF Fusion web applications will benefit most from the book. The book uses ADF business components as its model layer technology, ADF binding, ADF task flows and the ADF model for its controller layer technologies, and ADF Faces as its view layer technology.

The introductory concepts in the first chapter, along with the chapters related to handling exceptions, session timeouts, optimizing, and fine tuning may appeal more to application designers and architects.

Conventions

In this book, you will find a number of styles of text that distinguish between different kinds of information. Here are some examples of these styles, and an explanation of their meaning.

Code words in text are shown as follows: "In addition to the `session-timeout` configuration setting in `web.xml`, you can configure a session timeout warning interval by defining the context parameter"

A block of code is set as follows:

```
public class SessionTimeoutFilter implements Filter {
    private FilterConfig filterConfig = null;
    public SessionTimeoutFilter() {
        super();
    }
}
```

When we wish to draw your attention to a particular part of a code block, the relevant lines or items are set in bold:

```
new ExportEmployeesWork(getEmployees().createRowSetIterator(null))
```

Any command-line input or output is written as follows:

```
$ chmod u+x ./jdevstudio11121install.bin
$ ./jdevstudio11121install.bin
```

New terms and **important words** are shown in bold. Words that you see on the screen, in menus or dialog boxes for example, appear in the text like this: "Using the **Property Inspector** change the **URL Invoke** property to **url-invoke-allowed**."



Warnings or important notes appear in a box like this.



Tips and tricks appear like this.

Reader feedback

Feedback from our readers is always welcome. Let us know what you think about this book—what you liked or may have disliked. Reader feedback is important for us to develop titles that you really get the most out of.

To send us general feedback, simply send an e-mail to feedback@packtpub.com, and mention the book title through the subject of your message.

If there is a topic that you have expertise in and you are interested in either writing or contributing to a book, see our author guide on www.packtpub.com/authors.

Customer support

Now that you are the proud owner of a Packt book, we have a number of things to help you to get the most from your purchase.

Downloading the example code

You can download the example code files for all Packt books you have purchased from your account at <http://www.packtpub.com>. If you purchased this book elsewhere, you can visit <http://www.packtpub.com/support> and register to have the files e-mailed directly to you.

Errata

Although we have taken every care to ensure the accuracy of our content, mistakes do happen. If you find a mistake in one of our books—maybe a mistake in the text or the code—we would be grateful if you would report this to us. By doing so, you can save other readers from frustration and help us improve subsequent versions of this book. If you find any errata, please report them by visiting <http://www.packtpub.com/support>, selecting your book, clicking on the **errata submission form** link, and entering the details of your errata. Once your errata are verified, your submission will be accepted and the errata will be uploaded to our website, or added to any list of existing errata, under the Errata section of that title.

Piracy

Piracy of copyright material on the Internet is an ongoing problem across all media. At Packt, we take the protection of our copyright and licenses very seriously. If you come across any illegal copies of our works, in any form, on the Internet, please provide us with the location address or website name immediately so that we can pursue a remedy.

Please contact us at copyright@packtpub.com with a link to the suspected pirated material.

We appreciate your help in protecting our authors, and our ability to bring you valuable content.

Questions

You can contact us at questions@packtpub.com if you are having a problem with any aspect of the book, and we will do our best to address it.

1

Prerequisites to Success: ADF Project Setup and Foundations

In this chapter, we will cover:

- ▶ Installation of JDeveloper on Linux
- ▶ Breaking up the application in multiple workspaces
- ▶ Setting up BC base classes
- ▶ Setting up logging
- ▶ Using a custom exception class
- ▶ Using ADFUtils/JSFUtils
- ▶ Using page templates
- ▶ Using a generic backing bean actions framework

Introduction

JDeveloper and **ADF (Application Development Framework)** are amazing technologies. What makes them even more incredible is their sheer complexity and the amount of knowledge and effort that lies covered underneath the declarative, almost magical frontend. What amazes me is that once you scratch the surface, you never stop realizing how much you really don't know. Given this complexity, it becomes obvious that certain development guidelines and practices must be established and followed early in the architectural and design phases of an ADF project.

This chapter presents a number of recipes that are geared towards establishing some of these development practices. In particular, you will see content that serves as a starting point in making your own application modular when using the underlying technologies. You will also learn the importance of extending the Business Components framework (ADF-BC) base classes early in the development cycle. We will talk about the importance of laying out other application foundational components, such as logging and exceptions, again early in the development process, and continue with addressing reusability and consistency at the **ViewController** layer.

The chapter starts with a recipe about installing and configuring JDeveloper on Linux. So, let's get started and don't forget, have fun as you go along. If you get in trouble at any point, take a look at the accompanying source code and feel free to contact me anytime at nharalabidis@gmail.com.

Installation of JDeveloper on Linux

Installation of JDeveloper is, in general, a straightforward task. So, "why have a recipe for this?" you might ask. Did you notice the title? It says "on Linux". You will be amazed at the number of questions asked about this topic on a regular basis on the *JDeveloper and ADF OTN Forum*. Besides, in this recipe, we will also talk about configuration options and the usage of 64-bit JDK along with JDeveloper.

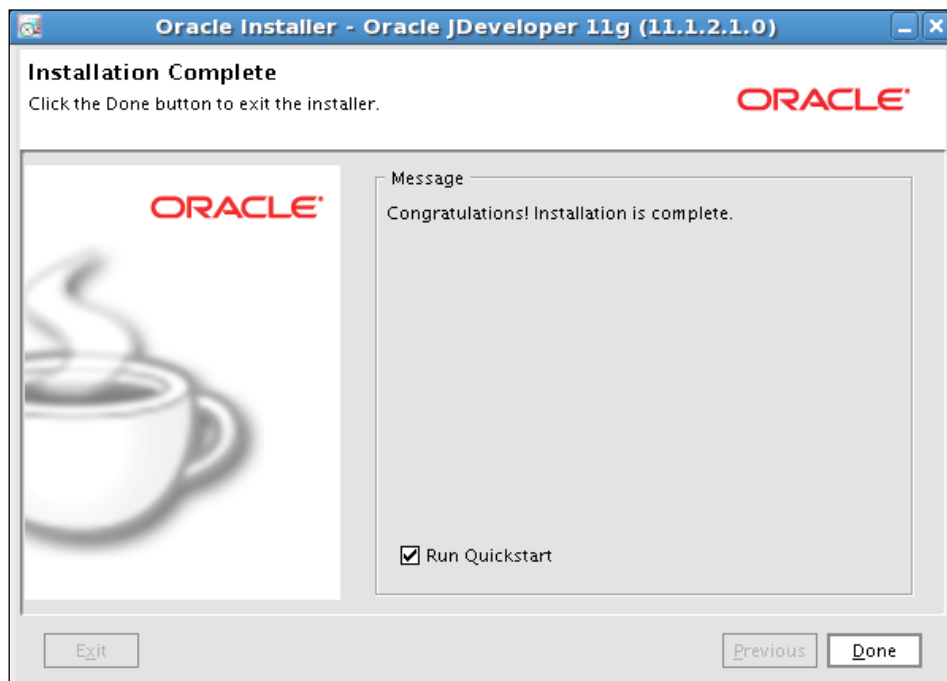
Getting ready

You will need a Linux installation of JDeveloper to use this recipe. For the 64-bit configuration, you will need a 64-bit Linux distribution and a 64-bit version of the Java SDK. We will install the latest version of JDeveloper, which is version 11.1.2.1.0 at the time of this writing.

How to do it...

1. Download JDeveloper from the Oracle JDeveloper Software download page:
<http://www.oracle.com/technetwork/developer-tools/jdev/downloads/index.html>.
2. Accept the license agreement, select **Linux Install**, and click on **Download File** to begin with the download.
3. Once the file is downloaded, open a console window and start the installation, by typing the following commands:

```
$ chmod u+x ./jdevstudio11121install.bin
$ ./jdevstudio11121install.bin
```
4. On the **Choose Middleware Home Directory** page, select **Create a new Middleware Home** and enter the Middleware home directory.
5. On the **Choose Install Type** page, select **Complete** to ensure that JDeveloper, ADF and WebLogic Server are installed.
6. Once you confirm your selections, proceed with the installation.
7. Upon a successful installation, you will see the **Installation Complete** page. Uncheck the **Run Quickstart** checkbox and click **Done**.



8. To start JDeveloper, go to the `/jdeveloper/jdev/bin` directory under the Middleware home directory you specified during the installation and type the following:

```
$ ./jdev
```

9. To make things easier, create an application launcher on your Linux desktop for the specific path indicated in the previous step.

How it works...

As noted earlier, installing JDeveloper on Linux is a straightforward task. You simply have to download the binary executable archive and run it. Ensure that you give execute permissions to the installation archive file and run it as noted. If you are having trouble seeing the **Welcome** page in graphical mode, ensure that the `$DISPLAY` environment variable is set correctly. The important thing to know here is the name of the file to execute in order to start JDeveloper. As mentioned, it is called `jdev` and it is located in the `/jdeveloper/jdev/bin` directory under the Middleware home directory.

There's more...

Now that you have successfully installed JDeveloper, let's spend some time configuring it for optimal performance. Configuration parameters are added to any of the `jdev.conf` or `ide.conf` files located in the `/jdeveloper/jdev/bin` and `/jdeveloper/ide/bin` directories respectively, under the Middleware home directory.

The following is a list of the important tuning configuration parameters with some recommendations for their values:

Parameter	Description
AddVMOption -Xmx	This parameter is defined in the <code>ide.conf</code> file and indicates the maximum limit that you will allow the JVM heap size to grow to. In plain words, it is the maximum memory that JDeveloper will consume on your system. When setting this parameter, consider the available memory on your system, the memory needed by the OS, the memory needed by other applications running concurrently with JDeveloper, and so on. On a machine used exclusively for development with JDeveloper, as a general rule of thumb consider setting it to around 50 percent of the available memory.

Parameter	Description
AddVMOption -Xms	This parameter is also defined in the <code>ide.conf</code> and indicates the initial JVM heap size. This is the amount that will be allocated initially by JDeveloper and it can grow up to the amount specified by the previous <code>-Xmx</code> parameter. When setting this parameter, consider whether you want to give JDeveloper a larger amount in order to minimize frequent adjustments to the JVM heap. Setting this parameter to the same value as the one indicated by the <code>-Xmx</code> parameter will supply a fixed amount of memory to JDeveloper.
AddVMOption -XX:MaxPermSize	This parameter indicates the size of the JVM permanent generation used to store class definitions and associated metadata. Increase this value if needed in order to avoid <code>java.lang.OutOfMemoryError: PermGen space</code> errors. A 256MB setting should suffice.
AddVMOption -DVFS_ ENABLE	Set it to <code>true</code> in <code>jdev.conf</code> if your JDeveloper projects consist of a large number of files, especially if you will be enabling a version control system from within JDeveloper.

Configuring JDeveloper with a 64-bit JDK

The JDeveloper installation is bundled by default with a 32-bit version of the Java JDK, which is installed along with JDeveloper. On a 64-bit system, consider running JDeveloper with a 64-bit version of the JDK. First download and install the 64-bit JDK. Then configure JDeveloper via the `SetJavaHome` configuration parameter in the `jdev.conf`. This parameter should be changed to point to the location of the 64-bit JDK. Note that the 64-bit JDK is supported by JDeveloper versions 11.1.1.4.0 and higher.

Configuring the JDeveloper user directory

This is the directory used by JDeveloper to identify a default location where files will be stored. JDeveloper also uses this location to create the integrated WebLogic domain and to deploy your web applications when running them or debugging them inside JDeveloper. It is configured via the `SetUserHomeVariable` parameter in the `jdev.conf` file. It can be set to a specific directory or to an environment variable usually named `JDEV_USER_DIR`. Note that when JDeveloper is started with the `-singleuser` command-line argument, the user directory is created inside the `/jdeveloper` directory under the Middleware home directory.



Before starting your development in JDeveloper, consider setting the XML file encoding for the XML files that you will be creating in JDeveloper. These files among others include, the JSF pages, the business component metadata files, application configuration files, and so on. You set the encoding via the **Tools | Preferences...** menu. Select the **Environment** node on the left of the **Preferences** dialog and the encoding from the **Encoding** drop-down. The recommended setting is **UTF-8** to support multi-lingual applications.



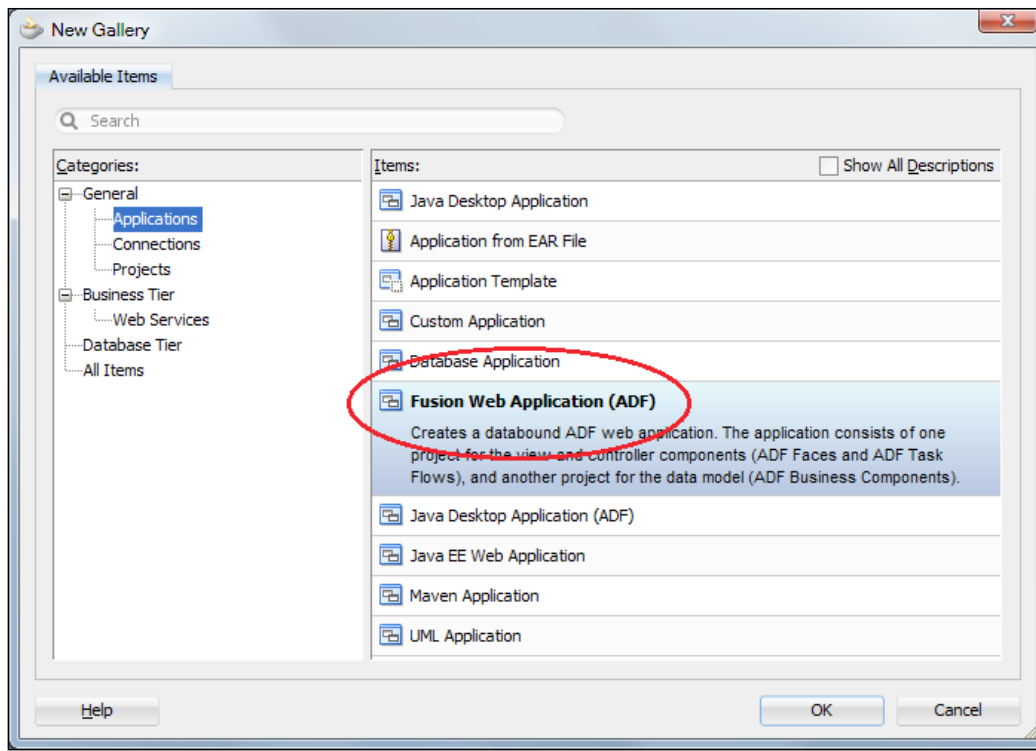
The minimum recommended open file descriptors limit for JDeveloper on a Linux system is 4096. Use the command `ulimit -n` to determine the open file descriptors limit for your installation and change it if needed in the `limits.conf` file located in `/etc/security/` directory.

Breaking up the application in multiple workspaces

When dealing with large enterprise scale applications, the organization and structure of the overall application in terms of JDeveloper workspaces, projects, and libraries is essential. Organizing and packaging ADF application artifacts, such as business components, task flows, templates, Java code, and so on, into libraries will promote and ensure modularity, and the reuse of these artifacts throughout the application. In this recipe, we will create an application that comprises reusable components. We will construct reusable libraries for shared components, business domain specific components, and a main application for consuming these components.

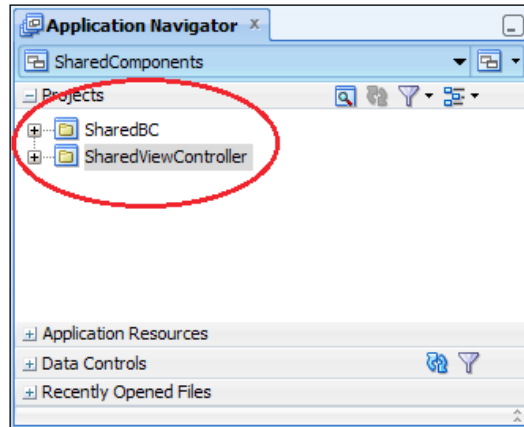
How to do it...

1. To create the `SharedComponents` library, start by selecting **New Application...** in the **Application Navigator**. This will start the application creation wizard.
2. In the **New Gallery** dialog, click on the **Applications** node (under the **General** category) and select **Fusion Web Application (ADF)** from the list of **Items**.

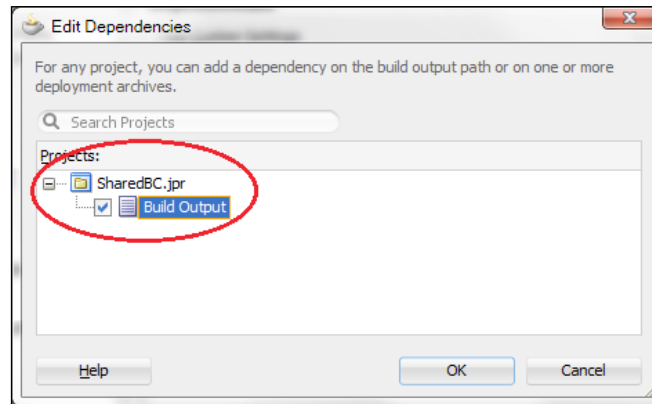


3. In the **Name your application** page, enter the **Application Name**, **Directory** and the **Application Package Prefix**.
4. In the **Name your project** page, enter the business component's **Project Name** and **Directory**. For this recipe, we have called it `SharedBC`.
5. In the **Configure Java settings** page for the business components project, accept the defaults for **Default Package**, **Java Source Path**, and **Output Directory**.
6. Similarly, in the **Name your project** page for the ViewController project, enter the **Project Name** and **Directory**. For this recipe, we have called the project `SharedViewController`. Ensuring that you enter a unique package structure for both projects is the best guarantee for avoiding naming conflicts when these projects are deployed as ADF Library JARs.
7. Accept the defaults in the **Configure Java settings** and click **Finish** to proceed with the creation of the workspace.

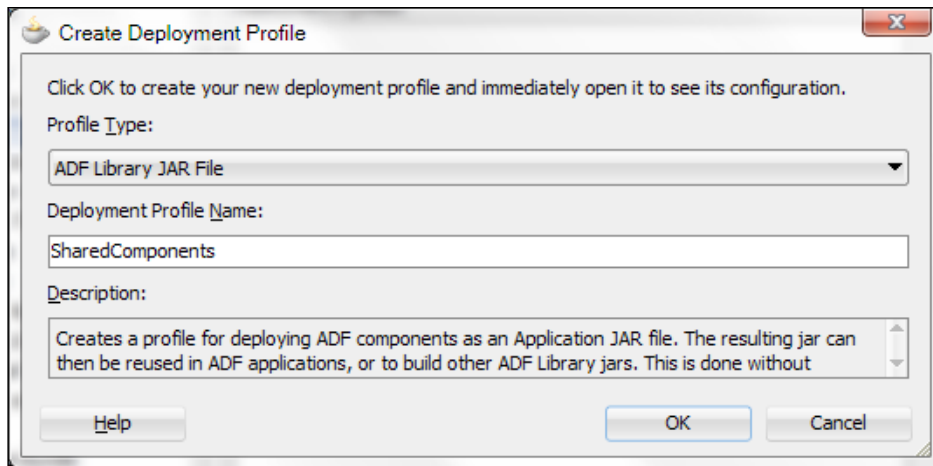
- Now, in the **Application Navigator**, you should see the two projects comprising the SharedComponents workspace, one for the business components and another for the ViewController.



- You will be using this workspace to add reusable business and ViewController components. For now, we will package the workspace into an ADF library JAR, without any components in it yet. In order to do this, you will need to first setup the project dependencies. Double-click on the SharedViewController project to bring up the **Project Properties** dialog and select **Dependencies**.
- Click on **Edit Dependencies** (the small pen icon) to bring up the **Edit Dependencies** dialog and then click on the **Build Output** checkbox under the business components project.
- Click **OK** to close the dialog and return to the **Project Properties** dialog.

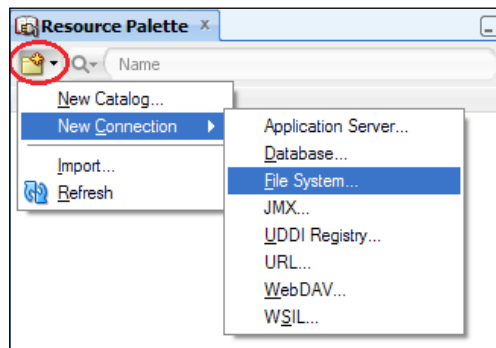


12. The next step is to set up the deployment profile. While at the **ViewController Project Properties** dialog, click on the **Deployment** node.
13. Since we will not be deploying this application as a WAR, select the default WAR deployment profile generated automatically by JDeveloper and delete it.
14. Then, click **New...** to create a new deployment profile.
15. On the **Create Deployment Profile** dialog, select **ADF Library JAR File** for the **Profile Type** and enter the name of the deployment profile. For this recipe, we have called the deployment profile `SharedComponents`. Click **OK** to proceed with its creation.

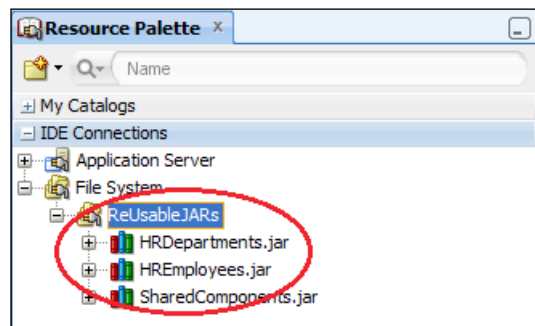


16. In the **Edit ADF Library JAR Deployment Profile Properties** dialog that is opened, select **JAR Options** and specify a location where you will be placing all the reusable JAR libraries. For this recipe, we will place all reusable libraries in a directory called `ReusableJARs`.
17. When done, completely exit from the **Project Properties** dialog, saving your changes by clicking **OK**.
18. The last step involves the creation of the **ADF Library JAR**. You do this by right-clicking on the `viewController` project in the **Application Navigator** selecting **Deploy** and then the name of the deployment profile name (`SharedComponents` in this case).
19. Select **Deploy to ADF Library JAR file** in the **Deployment Action** page and click **Finish** to initiate the deployment process. The deployment progress will begin. Its status is shown in the **Deployment** tab of the **Log** window.
20. To create the `HRDepartments` components library, similarly create a new Fusion web application for the `HRDepartment` components. Follow the previous steps to setup the project dependencies. No database connection to the `HR` schema is needed at this stage.

21. Create the deployment profile and deploy the ADF Library JAR. We will not be placing any components yet in this library.
22. To create the `HREmployees` components library, repeat the previous steps once more in order to create another ADF Library JAR for the `HR Employee` related reusable components.
23. Now create another Fusion web application, which will be used as the main application. This application will consume any of the components that reside in the ADF Library JARs created in the previous steps.
24. This can easily be done via the **Resource Palette** by creating a file system connection to the directory where we saved the reusable ADF Library JARs, that is, the directory called `ReUsableJARs`. If the **Resource Palette** is not visible, select **View | Resource Palette** to show it. In the **Resource Palette**, click on the **New** icon and select **New Connection | File System....**



25. In the **Create File System Connection** dialog that is displayed, enter the name of the connection and the directory where you have deployed the reusable components in the previous steps.
26. Click **OK** to continue. You should be able to see the new **File System Connection** in the **Resource Palette**.



27. To consume reusable components, first select the appropriate project on the **Application Navigator**, then right-click on the ADF Library JAR on the **Resource Palette** and select **Add to Project...**
28. On the **Confirm Add ADF Library** dialog, click on the **Add Library** button to proceed.
29. Alternatively, expand the ADF Library JAR and drag-and-drop the reusable component onto its appropriate place in the workspace.

How it works...

When you deploy a project as an ADF Library JAR, all ADF reusable components and code are packaged in it and they become available to other consuming applications and libraries. Reusable components include business components, database connections, data controls, task flows, task flow templates, page templates, declarative components, and of course Java code. By setting up the dependencies among the business components and ViewController projects in the way that we have—that is, including the build output of the business components project during the deployment of the ViewController project—you will be producing a single ADF Library JAR file with all the components from all the projects in the workspace. When you add an ADF Library JAR to your project, the library is added to the project's class path. The consuming project can then use any of the components in library. The same happens when you drag-and-drop a reusable component into your project.

There's more...

For this recipe, we packaged both of the business components and ViewController projects in the same ADF Library JAR. If this strategy is not working for you, you have other options, such as adjusting the dependencies among the two and packaging each project in a separate ADF Library JAR. In this case, you will need an additional deployment profile and a separate deployment for the business components project.

Adding the ADF Library JAR manually

You can add an ADF Library JAR into your project manually using the **Project Properties** dialog. Select the **Libraries and Classpath** node and click on the **Add Library...** button. This will display the **Add Library** dialog. On it, click the **New...** button to display the **Create Library** dialog. Enter a name for the library, select **Project** for the library location, and click on the **Deployed by Default** check button. Finally, click on the **Add Entry...** button to locate the ADF Library JAR. The **Deployed by Default** checkbox when checked indicates that the library will be copied to the application's destination archive during deployment of the consuming application. If you leave it unchecked, then the library will not be copied and it must be located in some other way (for example, deployed separately as a shared library on the application server).

Defining the application module granularity

One related topic that also needs to be addressed in the early architectural stages of the ADF project is the granularity for the application modules, that is, how the data model will be divided into application modules. As a general rule of thumb, each application module should satisfy a particular use case. Related use cases and, therefore, application modules can then be packaged into the same reusable ADF Library JAR. In general, avoid creating monolithic application modules that satisfy multiple use cases each.

Entity objects, list of values (LOVs), validation queries

Entity objects, list of values (LOVs) and validation queries should be defined only once for each business components project. To avoid duplication of entity objects, LOVs and validation queries among multiple business components projects, consider defining them only once in a separate business components project.



Structuring of the overall ADF application in reusable components should be well thought and incorporated in the early design and architectural phases of the project.

As your application grows, it is important to watch out for and eliminate circular dependencies among the reusable components that you develop. When they occur, this could indicate a flaw in your design. Use available dependency analyzer tools, such as Dependency Finder (available from <http://depfind.sourceforge.net>) during the development process, to detect and eliminate any circular dependencies that may occur.

Setting up BC base classes

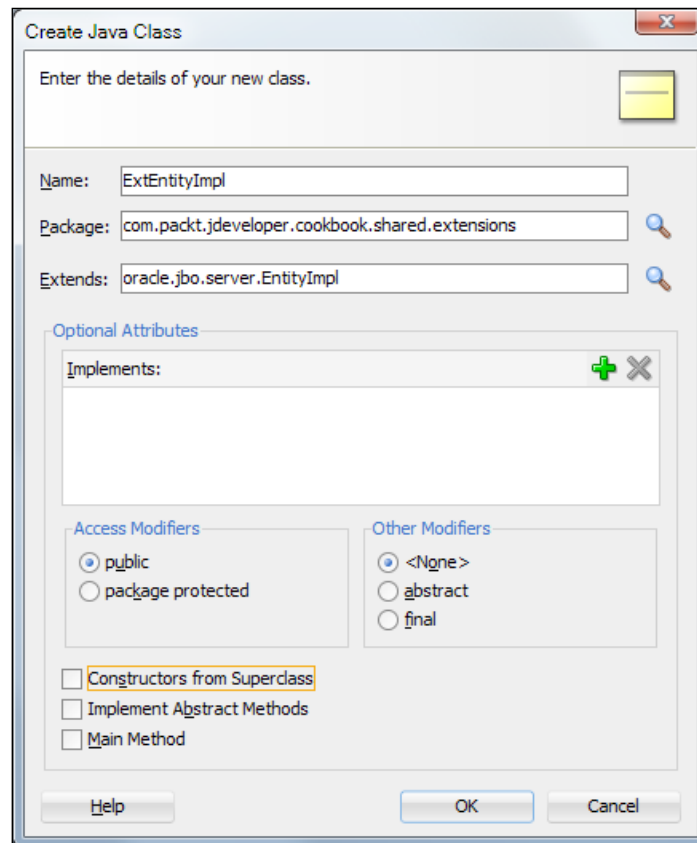
One of the first things to consider when developing large-scale enterprise applications with ADF-BC is to allow for the ability to extend the framework's base classes early on in the development process. It is imperative that you do this before creating any of your business objects, even though you have no practical use of the extended framework classes at that moment. This will guarantee that all of your business objects are correctly derived from your framework classes. In this recipe, you will expand on the previous recipe and add business components framework extension classes to the `SharedComponents` workspace.

Getting ready

You will be adding the business components framework extension classes to the `SharedComponents` workspace. See the previous recipe for information on how to create one.

How to do it...

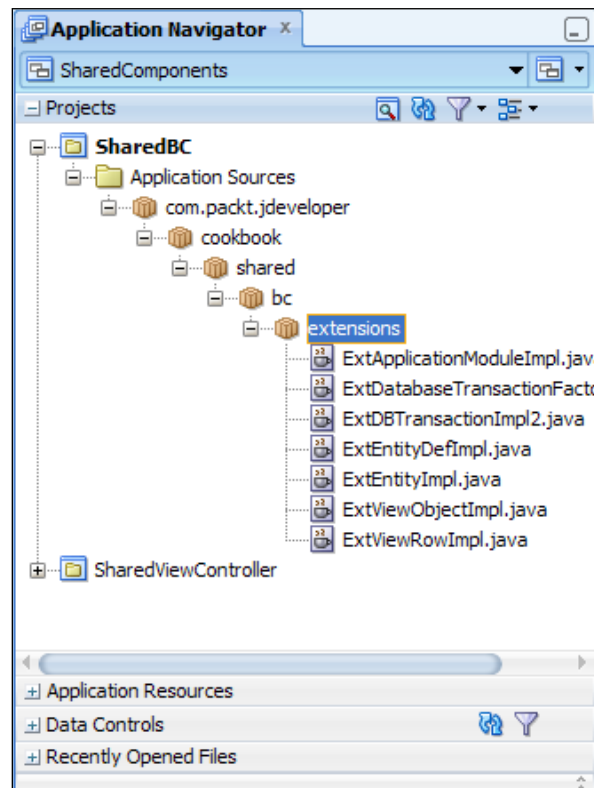
1. To create framework extension classes for the commonly used business components, start with the creation of an extension class for the entity objects. Open the `SharedComponents` workspace in JDeveloper and right-click on the `SharedBC` business components project.
2. From the context menu, select **New...** to bring up the **New Gallery** dialog. Select **Class** from the **Java** category (under the **General** category) and click **OK**.
3. On the **Create Java Class** dialog that is displayed, enter the name of the custom entity object class, the package where it will be created, and for **Extends** enter the base framework class, which in this case is `oracle.jbo.server.EntityImpl`.



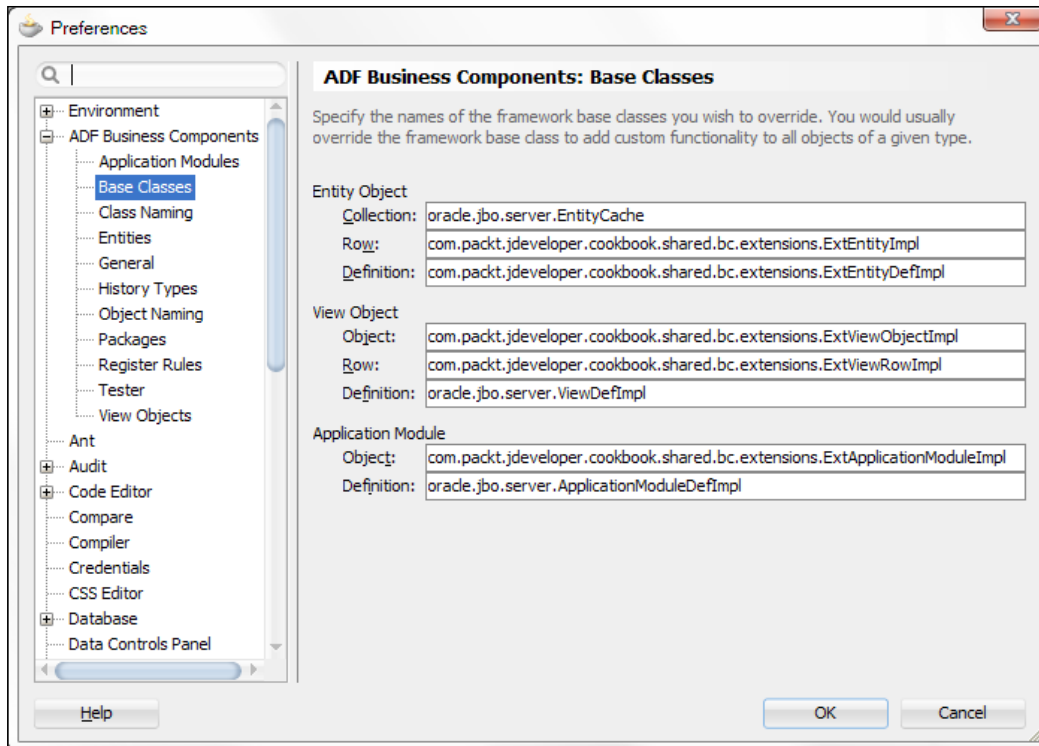
- Now, repeat the same steps to create framework extension classes for the following components:

Business Component	Framework Class Extended
Entity Definition	oracle.jbo.server.EntityDefImpl
View Object	oracle.jbo.server.ViewObjectImpl
View Row	oracle.jbo.server.ViewRowImpl
Application Module	oracle.jbo.server.ApplicationModuleImpl
Database Transaction Factory	oracle.jbo.server. DatabaseTransactionFactory
Database Transaction	oracle.jbo.server.DBTransactionImpl2

- Once you are done, your project should look similar to the following:



6. The next step is to configure JDeveloper so that all new business components that you create from this point forward will be inherited from the framework extension classes you've just defined. Open the **Preferences** dialog from the **Tools** menu, expand the **ADF Business Components** node, and select **Base Classes**.
7. Then enter the framework extension classes that you created previously, each one in its corresponding category.



How it works...

Defining and globally configuring business components framework extension classes via the **ADF Business Components Base Classes** settings on the **Preferences** dialog causes all subsequent business components for all projects to be inherited from these classes. This is true for both XML-only components and for components with custom Java implementation classes. For XML-only components observe that the `ComponentClass` attribute in the object's XML definition file points to your framework extension class.

There's more...

You can configure your business components framework extension classes at two additional levels: the project level and the individual component level.

- ▶ Configuration at the project level is done via the **Project Properties Base Classes** selection under the **ADF Business Components** node. These configuration changes will affect only the components created for the specific project.
- ▶ Configuration at the component level is done via the component's **Java Options** dialog, in the component's definition **Java** page, by clicking on the **Classes Extend...** button and overriding the default settings. The changes will only affect the specific component.



Do not attempt to directly change or remove the `extends` Java keyword in your component's implementation class. This would only be half the change, because the component's XML definition will still point to the original class. Instead, use the **Classes Extend...** button on the component's **Java Options** dialog.

Finally, note that the default package structure for all business components can also be specified in the **ADF Business Components | Packages** page of the **Preferences** dialog.

See also

- ▶ *Creating and using generic extension interfaces, Chapter 5, Putting them all together: Application Modules*
- ▶ *Breaking up the application in multiple workspaces, in this chapter*

Setting up logging

Logging is one of those areas that is often neglected during the initial phases of application design. There are a number of logging framework choices to use in your application, such as log4j by Apache. In this recipe, we will demonstrate the usage of the `ADFLogger` and Oracle Diagnostics Logging (ODL). The main advantage of using ODL when compared to other logging frameworks is its tight integration with WebLogic and JDeveloper. In WebLogic, the logs produced conform to and integrate with the diagnostics logging facility. Diagnostic logs include, in addition to the message logged, additional information such as the session and user that produced the log entry at run-time. This is essential when analyzing the application logs. In JDeveloper, the log configuration and analysis is integrated via the **Oracle Diagnostics Logging Configuration** and **Oracle Diagnostics Log Analyzer** respectively.

Getting ready

We will be adding logging to the application module framework extension class that we developed in the previous recipe.

How to do it...

1. ODL logs can be generated programmatically from within your code by using the `ADFLogger` class. Instantiate an `ADFLogger` via the static `createADFLogger()` method and use its `log()` method. Go ahead and add logging support to the application module framework extension class we developed in the previous recipe, as shown in the following code snippet:

```
import oracle.adf.share.logging.ADFLogger;
public class ExtApplicationModuleImpl extends
    ApplicationModuleImpl {
    // create an ADFLogger
    private static final ADFLogger LOGGER =
        ADFLogger.createADFLogger(ExtApplicationModuleImpl.class);
    public ExtApplicationModuleImpl() {
        super();
        // log a trace
        LOGGER.log(ADFLogger.TRACE,
            "ExtApplicationModuleImpl was constructed");
    }
}
```

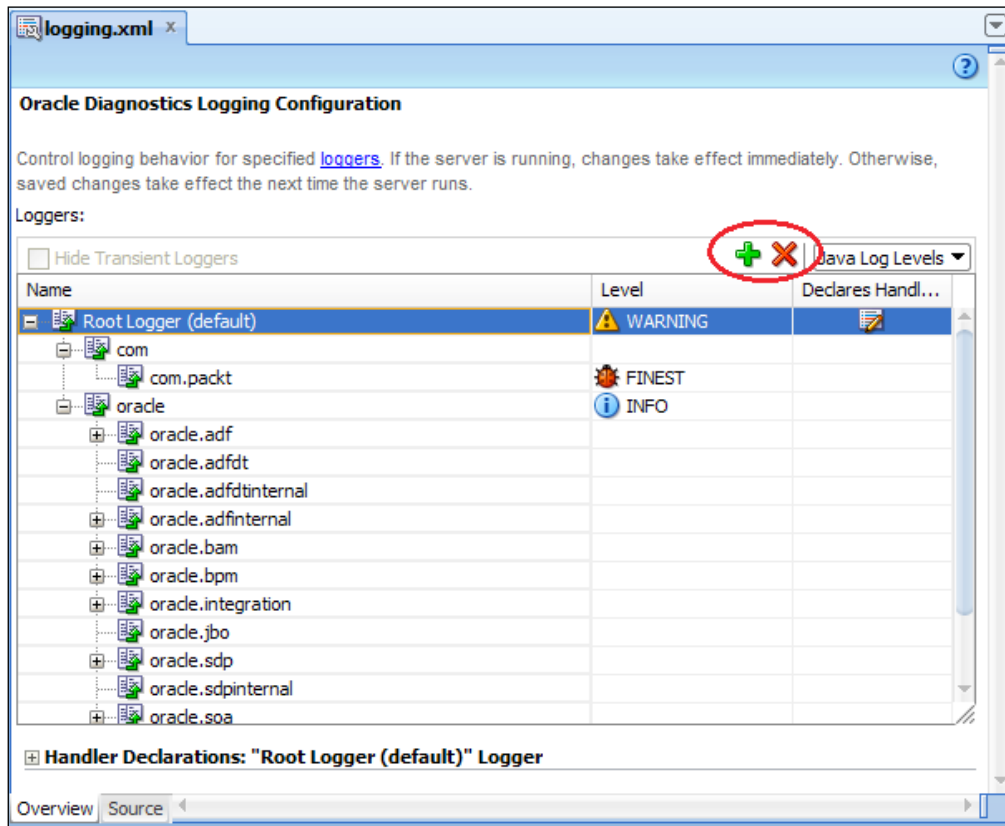


Downloading the example code

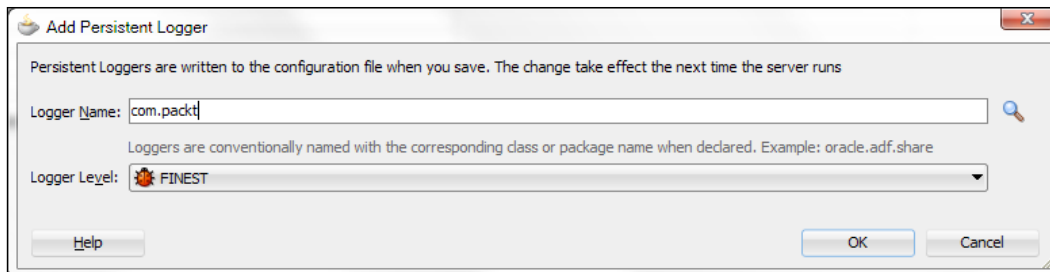
You can download the example code files for all Packt books you have purchased from your account at <http://www.packtpub.com>. If you purchased this book elsewhere, you can visit <http://www.packtpub.com/support> and register to have the files e-mailed directly to you.

2. The next step involves the configuration of the logger in the `logging.xml` file. The file is located in the `config\fmwconfig\servers` directory under the WebLogic domain for the server you are configuring. For the integrated WebLogic server, this file is located in the `%JDEV_USER_DIR%\system11.1.2.1.38.60.81\DefaultDomain\config\fmwconfig\servers\DefaultServer` directory. The exact location can vary slightly depending on the version of JDeveloper that you use.

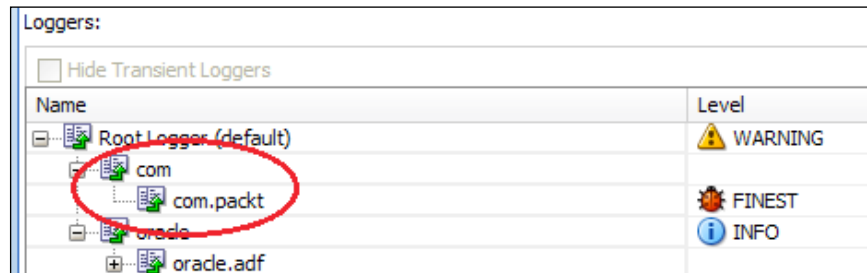
Open the file in JDeveloper and create a custom logger called `com.packt` by clicking on the **Add Persistent Logger** icon, as shown in the following screenshot:



3. This will display the **Add Persistent Logger** dialog to add your logger. Enter `com.packt` for the **Logger Name** and choose **FINEST** for the **Logger Level**.



- Repeat this step and add another logger named `com` if one does not already exist for it. The final result should look similar to the following screenshot:



- One more step that is required to complete the configuration is to use the `-Djbo.debugoutput=adflogger` and `-Djbo.adflogger.level=FINEST` options when starting the JVM. You can do this in JDeveloper by double-clicking on the main application's ViewController project to bring up the **Project Properties** dialog and selecting the **Run/Debug/Profile** node.
- Then select the appropriate **Run Configuration** on the right and click on the **Edit...** button.
- On the **Edit Run Configuration** dialog that is displayed, enter these Java options in the **Java Options**.

How it works...

In this example, we have declared a `static ADFLogger` and associated it with the class `ExtApplicationModuleImpl` by passing `ExtApplicationModuleImpl.class` as a parameter during its construction. We have declared the `ADFLogger` as `static` so we don't have to worry about passivating it. We then use its `log()` method to do our logging. The `log()` method accepts a `java.util.logging.Level` parameter indicating the log level of the message and it can be any of the following values: `ADFLogger.INTERNAL_ERROR`, `ADFLogger.ERROR`, `ADFLogger.WARNING`, `ADFLogger.NOTIFICATION`, or `ADFLogger.TRACE`.

`ADFLogger` leverages the Java Logging API to provide logging functionality. Because standard Java logging is used, it can be configured through the `logging.xml` configuration file. This file is located under the WebLogic domain directory `config\fmwconfig\servers` for the specific server that you are configuring. The file is opened and a logger is added.

Logging is controlled at the package level; we have added a logger for the `com.packt` package but we can fine-tune it for the additional levels: `com.packt.jdeveloper`, `com.packt.jdeveloper.cookbook`, `com.packt.jdeveloper.cookbook.shared`, and so on. The class name that we passed as an argument to the `ADFLogger` during its instantiation—that is, `ExtApplicationModuleImpl.class`—represents a logger that is defined in the logging configuration file. The logger that is added is a persistent logger, which means that it will remain permanently in the `logging.xml` configuration file. Transient loggers are also available; these persist only for the duration of the user session.

Each logger configured in the `logging.xml` is associated with a log handler. There are a number of handlers defined in the `logging.xml` namely a `console-handler` to handle logging to the console, an `odl_handler` to handle logging for ODL and others.

There's more...



You can also use the `ADFLogger` methods `severe()`, `warning()`, `info()`, `config()`, `fine()`, `finer()`, and `finest()` to do your logging.

When you configure logging, ensure that you make the changes to the appropriate `logging.xml` file for the WebLogic server you are configuring.

See also

- ▶ *Breaking up the application in multiple workspaces*, in this chapter
- ▶ *Configuring diagnostics logging*, Chapter 11, *Refactoring, Debugging, Profiling, Testing*
- ▶ *Dynamically configure ADF trace logs on WebLogic*, Chapter 11, *Refactoring, Debugging, Profiling, Testing*

Using a custom exception class

In this recipe, we will go over the steps necessary to set up a custom application exception class derived from the `JboException` base exception class. Some Reasons why you might want to do this include:

- ▶ Customize the exception error message
- ▶ Use error codes to locate the error messages in the resource bundle
- ▶ Use a single resource bundle per locale for the error messages and their parameters

Getting ready

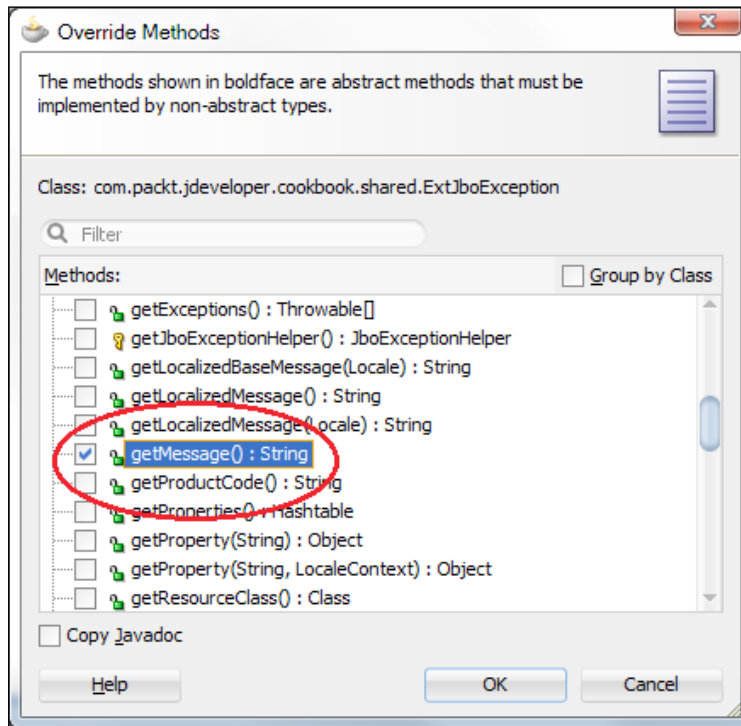
We will add the custom application exception class to the `SharedComponents` workspace we created in the *Breaking up the application in multiple workspaces* recipe in this chapter.

How to do it...

1. Start by opening the `SharedComponents` workspace.
2. Create a new class called `ExtJboException` by right-clicking on the business components project and selecting **New...**
3. Then select **Java** under the **General** category and **Java Class** from list of **Items** on the right.
4. Click **OK** to display the **Create Java Class** dialog. Enter `ExtJboException` for the **Name**, `com.packt.jdeveloper.cookbook.shared.bc.exceptions` for the **Package** and `oracle.jbo.JboException` for the **Extends**.
5. Click **OK** to proceed with the creation of the custom exception class.
6. The next step is to add two additional constructors, to allow for the instantiation of the custom application exception using a standard error message code with optional error message parameters. The additional constructors look similar to the following code sample:

```
public ExtJboException(final String errorCode,
    final Object[] errorParameters) {
    super(ResourceBundle.class, errorCode, errorParameters);
}
public ExtJboException(final String errorCode) {
    super(ResourceBundle.class, errorCode, null);
}
```

- Now, click on the **Override Methods...** icon on the top of the editor window and override the `getMessage()` method, as shown in the following screenshot:



- Enter the following code for the `getMessage()` method:

```
public String getMessage() {
    // default message
    String errorMessage = "";
    try {
        // get access to the error messages bundle
        final ResourceBundle messagesBundle = ResourceBundle.getBundle
            (ERRORS_BUNDLE, Locale.getDefault());
        // construct the error message
        errorMessage =this.getErrorCode() + " - " + messages
            Bundle.getString(MESSAGE_PREFIX + this.getErrorCode());
        // get access to the error message parameters bundle
        final ResourceBundle parametersBundle = ResourceBundle
            .getBundle(PARAMETERS_BUNDLE, Locale.getDefault());
        // loop for all parameters
        for (int i = 0; i < this.getErrorParameters().length; i++) {
            // get parameter value
            final String parameterValue =
```

```

        parametersBundle.getString(PARAMETER_PREFIX +
            (String)this.getErrorParameters()[i]);
        // replace parameter placeholder in the error message string
        errorMessage = errorMessage.replaceAll
            ("\\{" + (i + 1) + "}", parameterValue);
    }
} catch (Exception e) {
    // log the exception
    LOGGER.warning(e);
}
return errorMessage;
}

```

9. Make sure that you also add the following constants:

```

private static final String ERRORS_BUNDLE = "com.packt.jdeveloper.
    cookbook.shared.bc.exceptions.messages.ErrorMessages";
private static final String PARAMETERS_BUNDLE = "com.packt.
    jdeveloper.cookbook.shared.bc.exceptions.messages.ErrorParams";
private static final String MESSAGE_PREFIX = "message.";
private static final String PARAMETER_PREFIX = "parameter.";
private static final ADFLogger LOGGER =ADFLogger
    .createADFLogger(ExtJboException.class);

```

10. For testing purposes add the following main() method:

```

// for testing purposes; remove or comment if not needed
public static void main(String[] args) {
    // throw a custom exception with error code "00001" and two
    parameters
    throw new ExtJboException("00001",
        new String[] { "FirstParameter", "SecondParameter" });
}

```

How it works...

We have created a custom exception at the ADF-BC level by overriding the `JboException` class. In order to use application-specific error codes, we have introduced two new constructors. Both of them accept the error code as a parameter. One of them also accepts the message error parameters.

```

public ExtJboException(final String errorCode,
    final Object[] errorParameters) {
    super(ResourceBundle.class, errorCode, errorParameters);
}

```

In our constructor, we call the base class' constructor and pass the message error code and parameters to it.

Then we override the `getMessage()` method in order to construct the exception message. In `getMessage()`, we first get access to the error messages resource bundle by calling `ResourceBundle.getBundle()` as shown in the following code snippet:

```
final ResourceBundle messagesBundle = ResourceBundle.getBundle(ERRORS_
BUNDLE, Locale.getDefault());
```

This method accepts the name of the resource bundle and the locale. For the name of the resource bundle, we pass the constant `ERRORS_BUNDLE`, which we define as `com.packt.jdeveloper.cookbook.shared.bc.exceptions.messages.ErrorMessages`. This is the `ErrorMessages.properties` file in the `com/packt/jdeveloper/cookbook/shared/bc/exceptions/messages` directory where we have added all of our messages. For the locale, we use the default locale by calling `Locale.getDefault()`.

Then we proceed by loading the error message from the bundle:

```
errorMessage = this.getErrorCode() + " - " + messagesBundle.
getString(MESSAGE_PREFIX + this.getErrorCode());
```

An error message definition in the messages resource bundle looks similar to the following:

```
message.00001=This is an error message that accepts two parameters.
The first parameter is '{1}'. The second parameter is '{2}'.
```

As you can see, we have added the string prefix `message.` to the actual error message code. How you form the error message identifiers in the resource bundle is up to you. You could, for example, use a module identifier for each message and change the code in `getMessage()` appropriately. Also, we have used braces, that is, `{1}`, `{2}` as placeholders for the actual message parameter values. Based on all these, we constructed the message identifier by adding the message prefix to the message error code as: `MESSAGE_PREFIX + this.getErrorCode()` and called `getString()` on the `messagesBundle` to load it.

Then we proceed with iterating the message parameters. In a similar fashion, we call `getString()` on the parameters bundle to load the parameter values.

The parameter definitions in the parameters resource bundle look similar to the following:

```
parameter.FirstParameter=Hello
parameter.SecondParameter=World
```

So we add the prefix `parameter` to the actual parameter identifier before loading it from the bundle.

The last step is to replace the parameter placeholders in the error message with the actual parameter values. We do this by calling `replaceAll()` on the raw error message, as shown in the following code snippet:

```
errorMessage = errorMessage.replaceAll("\\{" + (i + 1) + "}",  
parameterValue);
```

For testing purposes, we have added a `main()` method to test our custom exception. You will similarly throw the exception in your business components code, as follows:

```
throw new ExtJboException("00001", // message code  
    new String[] { "FirstParameter", "SecondParameter" }  
    // message parameters);
```

There's more...

You can combine the error message and the error message parameters bundles into a single resource bundle, if you want, and change the `getMessage()` method as needed to load both from the same resource bundle.

Bundled Exceptions

By default, exceptions are bundled at the transaction level for ADF-BC-based web applications. This means that all exceptions thrown during attribute and entity validations are saved and reported once the validation process is complete. In other words, the validation will not stop on the first error, rather it will continue until the validation process completes and then report all exceptions in a single error message. Bundled validation exceptions are implemented by wrapping exceptions as details of a new parent exception that contains them. For instance, if multiple attributes in a single entity object fail attribute validation, these multiple `ValidationException` objects are wrapped in a `RowValException`. This wrapping exception contains the row key of the row that has failed validation. At transaction commit time, if multiple rows do not successfully pass the validation performed during commit, then all of the `RowValException` objects will get wrapped in an enclosing `TxnValException` object. Then you can use the `getDetails()` method of the `JboException` base exception class to recursively process the bundled exceptions contained inside it.

Exception bundling can be configured at the transaction level by calling `setBundledExceptionMode()` on the `oracle.jbo.Transaction`. This method accepts a Boolean value indicating that bundled transactions will be used or not, respectively.



Note that in the *Using a generic backing bean actions framework* recipe in this chapter, we refactored the code in `getMessage()` to a reusable `BundleUtils.loadMessage()` method. Consequently, we changed the `ExtJboException` `getMessage()` in that recipe to the following:

```
public String getMessage() {
    return BundleUtils.loadMessage(this.getErrorCode(),
        this.getErrorParameters());
}
```

See also

- ▶ *Handling security, session timeouts, exceptions and errors*, Chapter 9, *Handling Security, Session Timeouts, Exceptions and Errors*
- ▶ *Breaking up the application in multiple workspaces*, in this chapter

Using ADFUtils/JSFUtils

In this recipe, we will talk about how to incorporate and use the `ADFUtils` and `JSFUtils` utility classes in your ADF application. These are utility classes used at the `ViewController` level that encapsulate a number of lower level ADF and JSF calls into higher level methods. Integrating these classes in your ADF application early in the development process, and subsequently using them, will be of great help to you as a developer and contribute to the overall project's clarity and consistency. The `ADFUtils` and `JSFUtils` utility classes, at the time of writing, are not part of any official JDeveloper release. You will have to locate them, configure them, and expand them as needed in your project.

Getting ready

We will be adding the `ADFUtils` and `JSFUtils` classes to the `SharedComponents` `ViewController` project that we developed in the *Breaking up the application in multiple workspaces* recipe in this chapter.

How to do it...

1. To get the latest version of these classes, download and extract the latest version of the **Fusion Order Demo** application in your PC. This sample application can be found currently in the **Fusion Order Demo (FOD) - Sample ADF Application** page at the following address: <http://www.oracle.com/technetwork/developer-tools/jdev/index-095536.html>.

2. The latest version of the Fusion Order Demo application is 11.1.2.1 R2 at the time of this writing and is bundled in a zipped file. So go ahead download and extract the Fusion Order Demo application in your PC.
3. You should be able to locate the `ADFUtils` and `JSFUtils` classes in the location where you have extracted the Fusion Order Demo application. If multiple versions of the same class are found, compare them and use the ones that are most up-to-date. For this recipe, we have included in the source code the `ADFUtils` and `JSFUtils` found in the `SupplierModule\ViewController\src\oracle\fodemo\supplier\view\utils` directory.
4. Copy these classes to a specific location in your shared `ViewController` components project. For this recipe, we have copied them into the `SharedComponents\SharedViewController\src\com\packt\jdeveloper\cookbook\shared\view\util` directory.
5. Once copied, open both files with `JDeveloper` and change their package to reflect their new location, in this case to `com.packt.jdeveloper.cookbook.shared.view.util`.

How it works...

The public interfaces of both `ADFUtils` and `JSFUtils` define `static` methods, so you can call them directly without any class instantiations. The following are some of the methods that are commonly used.

Locating an iterator binding

To locate an iterator in the bindings, use the `ADFUtils.findIterator()` method. The method accepts the bound iterator's identifier and returns an `oracle.adf.model.binding.DCIteratorBinding`. The following is an example:

```
DCIteratorBinding it = ADFUtils.findIterator("IteratorID");
```

Locating an operation binding

To locate an operation in the bindings, use the `ADFUtils.findOperation()` method. This method accepts the bound operation's identifier and returns an `oracle.binding.OperationBinding`.

```
OperationBinding oper = ADFUtils.findOperation("OperationID");
```


Locating an attribute binding

Use `ADFUtils.findControlBinding()` to retrieve an attribute from the bindings. This method accepts the bound attribute's identifier and returns an `oracle.binding.AttributeBinding`.

```
AttributeBinding attrib =
    ADFUtils.findControlBinding("AttributeId");
```

Getting and setting an attribute binding value

To get or set a bound attribute's value, use the `ADFUtils.getBoundAttributeValue()` and `ADFUtils.setBoundAttributeValue()` methods respectively. Both of these methods accept the identifier of the attribute binding as an argument. The `getBoundAttributeValue()` method returns the bound attribute's data value as a `java.lang.Object`. The `setBoundAttributeValue()` method accepts a `java.lang.Object` and uses it to set the bound attribute's value.

```
// get some bound attribute data
String someData =
    (String)ADFUtils.getBoundAttributeValue("AttributeId");
// set some bound attribute data
ADFUtils.setBoundAttributeValue("AttributeId", someData);
```

Getting the binding container

You can get the `oracle.adf.model.binding.DCBindingContainer` binding container by calling the `ADFUtils.getDCBindingContainer()` method.

```
DCBindingContainer bindings = ADFUtils.getDCBindingContainer();
```

Adding Faces messages

Use the `JSFUtils.addFacesInformationMessage()` and `JSFUtils.addFacesErrorMessage()` methods to display Faces information and error messages respectively. These methods accept the message to display as a `String` argument.

```
JSFUtils.addFacesInformationMessage("Information message");
JSFUtils.addFacesErrorMessage("Error message");
```

Finding a component in the root view

To locate a UI component in the root view based on the component's identifier, use the `JSFUtils.findComponentInRoot()` method. This method returns a `javax.faces.component.UIComponent` matching the specified component identifier.

```
UIComponent component = JSFUtils.findComponentInRoot("ComponentID");
```

Getting and setting managed bean values

Use the `JSFUtils.getManagedBeanValue()` and `JSFUtils.setManagedBeanValue()` methods to get and set a managed bean value respectively. These methods both accept the managed bean name. The `JSFUtils.getManagedBeanValue()` method returns the managed bean value as a `java.lang.Object`. The `JSFUtils.setManagedBeanValue()` method accepts a `java.lang.Object` and uses it to set the managed bean value.

```
Object filePath = JSFUtils.getManagedBeanValue
("bindings.FilePath.inputValue");
JSFUtils.setManagedBeanValue("bindings.FilePath.inputValue", null);
```

Using page templates

In this recipe, we will go over the steps required to create a JSF page template that you can use to create JSF pages throughout your application. It is very likely that for a large enterprise-scale application you will need to construct and use a number of different templates, each serving a specific purpose. Using templates to construct the actual application JSF pages will ensure that pages throughout the application are consistent, and provide a familiar look and feel to the end user. You can follow the steps presented in this recipe to construct your page templates and adapt them as needed to fit your own requirements.

Getting ready

We will be adding the JSF template to the `SharedComponents` ViewController project that we developed in the *Breaking up the application in multiple workspaces* recipe in this chapter.

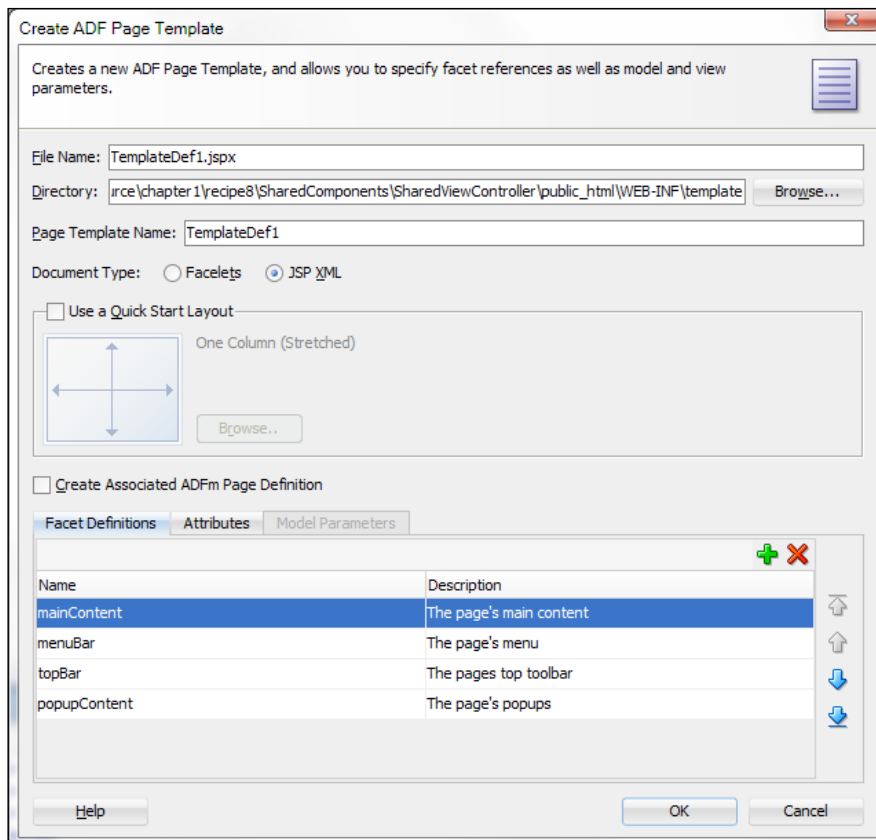
How to do it...

1. Start by right-clicking on the ViewController project in the `SharedComponents` workspace and selecting **New...**
2. On the **New Gallery** dialog select **JSF/Facelets** from the list of **Categories** and **ADF Page Template** from the **Items** on the right.
3. Click **OK** to proceed. This will display the **Create ADF Page Template** dialog.
4. Enter the name of the template on the **Page Template Name**. Note that as you change the template name, the template **File Name** also changes to reflect the template name. For this recipe, we will simply call the template `TemplateDef1`.
5. Now, click on the **Browse...** button and select the directory where the template will be stored.
6. On the **Choose Directory** dialog navigate to the `public_html/WEB-INF` directory and click on the **Create new subdirectory** icon to create a new directory called `templates`.

7. For the **Document Type**, select **JSP XML**.
8. We will not be using any of the pre-defined templates, so uncheck the **Use a Quick Start Layout** checkbox.
9. Also, since we will not be associating any data bindings to the template, uncheck the **Create Associated ADFm Page Definition** checkbox.
10. Next, you will be adding the template facets. You do this by selecting the **Facet Definitions** tab and clicking on the **New** icon button. Enter the following facets:

Facet	Description
mainContent	This facet will be used for the page's main content.
menuBar	This facet will be used to define a menu at the top of the page.
topBar	This facet will be used to define a toolbar under the page's menu.
popupContent	This facet will be used to define the page's pop-ups.

11. Now click **OK** to proceed with the creation of the ADF page template.



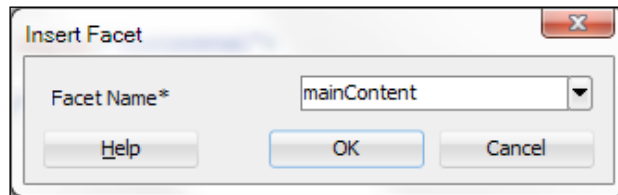
12. Once the template is created, it is opened in the JDeveloper editor. If you followed the previous steps, the template should look similar to the following code snippet:

```
<af:pageTemplateDef var="attrs">
  <af:xmlContent>
    <component xmlns
      ="http://xmlns.oracle.com/adf/faces/rich/component">
      <display-name>TemplateDef1</display-name>
      <facet>
        <description>The page's main content</description>
        <facet-name>mainContent</facet-name>
      </facet>
      <facet>
        <description>The page's menu</description>
        <facet-name>menuBar</facet-name>
      </facet>
      <facet>
        <description>The page's top toolbar</description>
        <facet-name>topBar</facet-name>
      </facet>
      <facet>
        <description>The page's popups</description>
        <facet-name>popupContent</facet-name>
      </facet>
    </component>
  </af:xmlContent>
</af:pageTemplateDef>
```

As you can see, at this point, the template contains only its definition in an `af:xmlContent` tag with no layout information whatsoever. We will proceed by adding the template's layout content.

13. From the **Layout** components in the **Component Palette**, grab a `Form` component and drop it into the template.
14. From the **Layout** container, grab a `Panel Stretch Layout` and drop it into the `Form` component. Remove the top, bottom, start, and end facets.
15. From the **Layout** container, grab a `Panel Splitter` component and drop it on the center facet of the `Panel Stretch Layout`. Using the **Property Inspector** change the `Panel Splitter Orientation` to vertical. Also adjust the `SplitterPosition` to around 100.
16. Add your application logo by dragging and dropping an `Image` component from the **General Controls** onto the first facet of the `Panel Splitter`. For this recipe, we have created a `public_html\images` directory and we copied a `logo.jpg` logo image there. We then specified `/images/logo.jpg` as image `Source` for the `Image` component.

17. Let's proceed by adding the main page's layout content. Drop a **Decorative Box** from the **Layout** components onto the second facet of the **Panel Splitter**. We will not be using the top facet of **Decorative Box**, so remove it.
18. OK, we are almost there! Drag a **Panel Stretch Layout** from the **Layout** components and drop it onto the center facet of the **Decorative Box**. Remove the start and end facets, since we will not be using them.
19. Drag a **Facet Ref** component from the **Layout components** and drop it onto the center facet of the **Panel Stretch Layout**. On the **Insert Facet** dialog, select the **mainContent** facet that you added during the template creation.



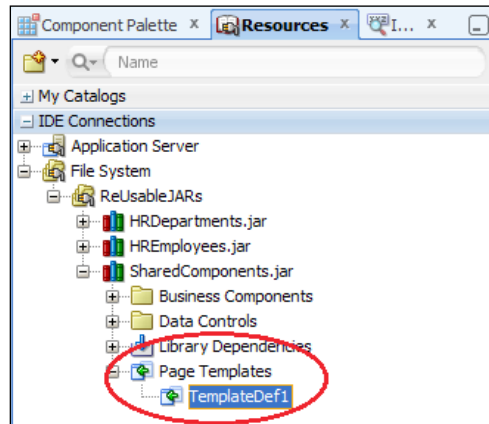
20. Finally, add the following code to the **Panel Stretch Layout topBar** facet:

```
<f:facet name="top">
  <af:panelGroupLayout id="pt_pgl5" layout="vertical">
    <af:facetRef facetName="popupContent"/>
    <af:menuBar id="pt_mb1">
      <af:facetRef facetName="menuBar"/>
    </af:menuBar>
    <af:panelGroupLayout id="pt_pgl2" layout="horizontal">
      <af:toolbar id="pt_t2">
        <af:facetRef facetName="topBar"/>
      </af:toolbar>
    </af:panelGroupLayout>
  </af:panelGroupLayout>
</f:facet>
```

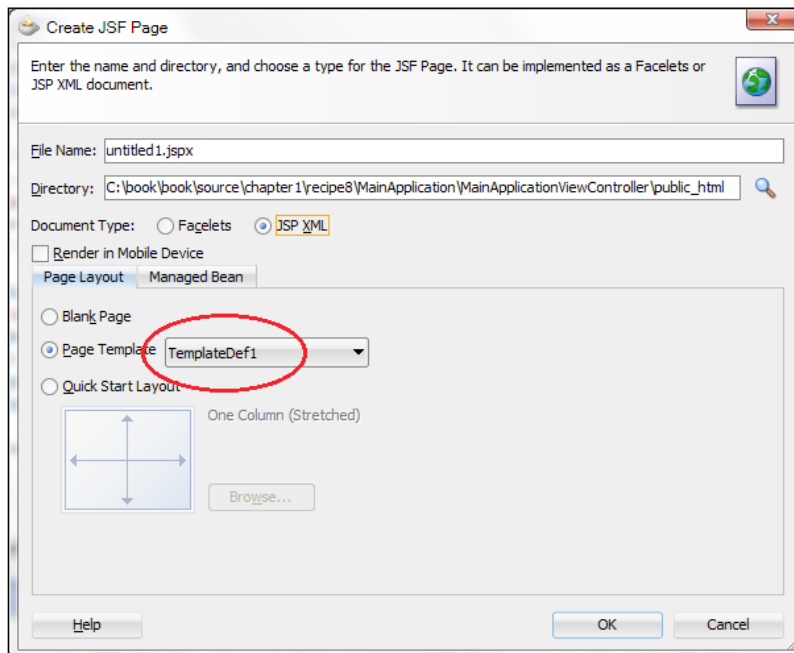
How it works...

When the template is created, there is no layout information in it, so we have to add it ourselves. We do this by using a variety of layout components to arrange the contained UI. Also, notice the usage of the `af:facetRef` component. It is being used to reference a template facet in the specific place within the layout content. The facet is then available to you when you create a JSF page from the template. This will become obvious when we generate a JSF page from the template. Note that each `Facet` can only be added once to the template.

So, how do you use the JSF page template? Since we have created the template in a `SharedComponents` project, we will first need to deploy the project to an ADF Library JAR. Then we will be able to use it from other consuming projects. This was explained in the *Breaking up the application in multiple workspaces* recipe, earlier in this chapter. When you do so, the template will be visible to all consuming projects, as shown in the following screenshot:



Once the ADF Library JAR containing the template is added to the consuming project, you can see and select the template when you create a new JSF page in the **Create JSF Page** dialog. The template introduced in this recipe is shown in the following screenshot:



The XML source code that is generated for a JSF page created from this template will look similar to the following code snippet:

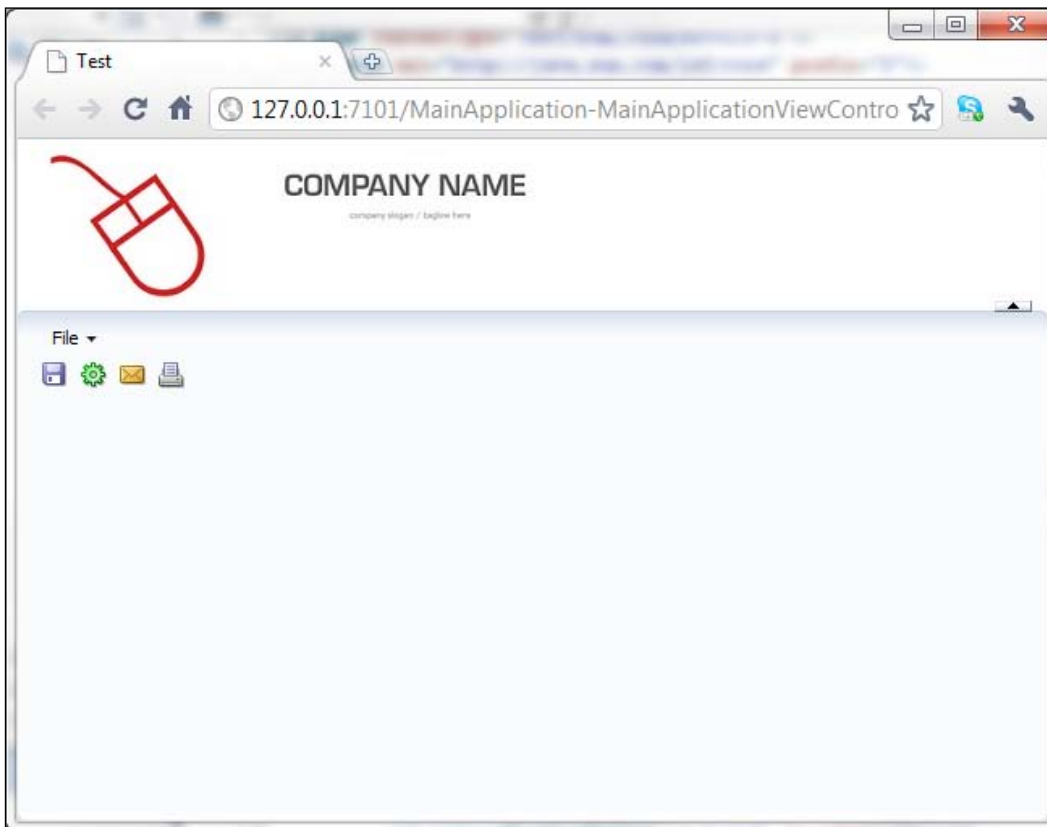
```
<f:view>
  <af:document id="d1" title="Test">
    <af:pageTemplate viewId="/WEB-INF/templates/TemplateDef1.jspx"
      id="pt1">
      <f:facet name="mainContent"/>
      <f:facet name="menuBar"/>
      <f:facet name="topBar"/>
      <f:facet name="bottomBar"/>
      <f:facet name="popupContent"/>
    </af:pageTemplate>
  </af:document>
</f:view>
```

You can see in the listing that the page references the template via the `af:pageTemplate` tag. The template facets that you have defined are available so you can enter the page-specific UI content. After adding an `af:menuBar` to the `menuBar` facet and some `af:commandToolBarButton` components to the `topBar` facet, the JSF page could look similar to the following code:

```
<f:view>
  <af:document id="d1" title="Test">
    <af:pageTemplate viewId="/WEB-INF/templates/TemplateDef1.jspx"
      id="pt1">
      <f:facet name="mainContent"/>
      <f:facet name="menuBar">
        <af:menuBar id="mb1">
          <af:menu text="File" id="m1">
            <af:commandMenuItem text="Save" id="cmi1"
              icon="/images/filesave.png"/>
            <af:commandMenuItem text="Action" id="cmi2"
              icon="/images/action.png"/>
            <af:commandMenuItem text="Mail" id="cmi3"
              icon="/images/envelope.png"/>
            <af:commandMenuItem text="Print" id="cmi4"
              icon="/images/print.png"/>
          </af:menu>
        </af:menuBar>
      </f:facet>
      <f:facet name="topBar">
        <af:group id="g1">
          <af:commandToolBarButton id="ctb1" shortDesc="Save"
            icon="/images/filesave.png"/>
        </af:group>
      </f:facet>
    </af:pageTemplate>
  </af:document>
</f:view>
```

```
<af:commandToolBarButton id="ctb2" shortDesc="Action"
    icon="/images/action.png"/>
<af:commandToolBarButton id="ctb3" shortDesc="Mail"
    icon="/images/envelope.png"/>
<af:commandToolBarButton id="ctb4" shortDesc="Print"
    icon="/images/print.png"/>
</af:group>
</f:facet>
<f:facet name="popupContent"/>
</af:pageTemplate>
</af:document>
</f:view>
```

Running the page in JDeveloper will produce the following:



There's more...

Although adding a `Form` component to a template is not recommended practice, this is not a problem for the template created in this recipe, since we will not be using it for the creation of page fragments. Using a template that contains a `Form` component to create page fragments will result in a problem when a consuming page already contains a `Form` component itself. The template developed in this recipe will not be used for page fragments. It was developed specifically to be used along with the generic backing bean actions framework explained in the *Using a generic backing bean actions framework* recipe in this chapter.

Using a generic backing bean actions framework

In this recipe we will create a base backing bean class that we will use to encapsulate common functionality for common JSF page actions, such as committing and rolling back data, creating new records, deleting records and so on. Creating and using such a generic backing bean actions framework will guarantee that you provide consistent functionality throughout the application and encapsulate common functionality at a base class level. This class is not intended to be used as a utility class. Any new helper methods that were developed to demonstrate the recipe were added to the `ADFUtils` utility class discussed earlier in this chapter.

Getting ready

We will be adding the generic backing bean actions framework to the `SharedComponents` `ViewController` project that we developed in the *Breaking up the application in multiple workspaces* recipe in this chapter.

How to do it...

1. Right-click on the shared `ViewController` project and select **New...**
2. On the **New Gallery** dialog, select **Java** under the **General** category and **Java Class** from the list of items on the right.
3. On the **Create Java Class** dialog, enter `CommonActions` for the class name and `com.packt.jdeveloper.cookbook.shared.view.actions` for the class package.
4. Let's go ahead and add methods to provide consistent commit functionality:

```
public void commit(ActionEvent actionEvent) {  
    if (ADFUtils.hasChanges()) {  
        // allow derived beans to handle before commit actions  
        onBeforeCommit(actionEvent);  
    }  
}
```

```

        // allow derived beans to handle commit actions
        onCommit(actionEvent);
        // allow derived beans to handle after commit actions
        onAfterCommit(actionEvent);
    } else {
        // display "No changes to commit" message
        JSFUtils.addFacesInformationMessage(BundleUtils.
            loadMessage("00002"));
    }
}
protected void onBeforeCommit(ActionEvent actionEvent) {
}
/**
protected void onCommit(ActionEvent actionEvent) {
    // execute commit
    ADFUtils.execOperation(Operations.COMMIT);
}
protected void onAfterCommit(ActionEvent actionEvent) {
    // display "Changes were committed successfully" message
    JSFUtils.addFacesInformationMessage(BundleUtils.
        loadMessage("00003"));
}

```

5. We have also added similar methods for consistent rollback behaviour. To provide uniform record creation/insertion functionality, let's add these methods:

```

public void create(ActionEvent actionEvent) {
    if (hasChanges()) {
        onCreatePendingChanges(actionEvent);
    } else {
        onContinueCreate(actionEvent);
    }
}
protected void onBeforeCreate(ActionEvent actionEvent) {
    // commit before creating a new record
    ADFUtils.execOperation(Operations.COMMIT);
}
public void onCreate(ActionEvent actionEvent) {
    execOperation(Operations.INSERT);
}
protected void onAfterCreate(ActionEvent actionEvent) {
}
public void onCreatePendingChanges(ActionEvent actionEvent) {
    ADFUtils.showPopup("CreatePendingChanges");
}
public void onContinueCreate(ActionEvent actionEvent) {
    onBeforeCreate(actionEvent);
    onCreate(actionEvent);
    onAfterCreate(actionEvent);
}

```

6. Similar methods were added for consistent record deletion behaviour. In this case, we have added functionality to show a delete confirmation pop-up.

How it works...

To provide consistent functionality at the JSF page actions level, we have implemented the `commit()`, `rollback()`, `create()`, and `remove()` methods. Derived backing beans should handle these actions by simply delegating to this base class via calls to `super.commit()`, `super.rollback()`, and so on. The base class `commit()` implementation first calls the helper `ADFUtils.hasChanges()` to determine whether there are transaction changes. If there are, then the `onBeforeCommit()` is called to allow derived backing beans to perform any pre-commit processing. Commit processing continues by calling `onCommit()`. Again, derived backing beans can override this method to provide specialized commit processing. The base class implementation of `onCommit()` calls the helper `ADFUtils.execOperation()` to execute the `Operations.COMMIT` bound operation. The commit processing finishes by calling the `onAfterCommit()`. Derived backing beans can override this method to perform post-commit processing. The default base class implementation displays a **Changes were committed successfully** message on the screen.

The generic functionality for a new record creation is implemented in the `create()` method. Derived backing beans should delegate to this method for default record creation processing by calling `super.create()`. In `create()`, we first check to see if we have any changes to the existing transaction. If we do, we will inform the user by displaying a message dialog. We do this in the `onCreatePendingChanges()` method. The default implementation of this method displays the `CreatePendingChanges` confirmation pop-up. The derived backing bean can override this method to handle this event in a different manner. If the user chooses to go ahead with the record creation, the `onContinueCreate()` is called. This method calls `onBeforeCreate()` to handle precreate functionality. The default implementation commits the current record by calling `ADFUtils.execOperation(Operations.COMMIT)`. Record creation continues with calling `onCreate()`. The default implementation of this method creates and inserts the new record by calling `ADFUtils.execOperation(Operations.INSERT)`. Finally, `onAfterCreate()` is called to handle any creation post processing.

The generic rollback and record deletion functionality is similar. For the default delete processing, a pop-up is displayed asking the user to confirm whether the record should be deleted or not. The record is deleted only after the user's confirmation.

There's more...

Note that this framework uses a number of pop-ups in order to confirm certain user choices. Rather than adding these pop-ups to all JSF pages, these pop-ups are added once to your JSF page template, providing reusable pop-ups for all of your JSF pages. In order to support this generic functionality, additional plumbing code will need to be added to the actions framework. We will talk at length about it in the *Using page templates for pop-up reuse* recipe in *Chapter 7, Face Value: ADF Faces, JSPX Pages and Components*.

See also

- ▶ *Using page templates for pop-up reuse, Chapter 7, Face Value: ADF Faces, JSPX Pages and Components*
- ▶ *Breaking up the application in multiple workspaces, in this chapter*

2

Dealing with Basics: Entity Objects

In this chapter, we will cover:

- ▶ Using a custom property to populate a sequence attribute
- ▶ Overriding doDML() to populate an attribute with a gapless sequence
- ▶ Creating and applying property sets
- ▶ Using getPostedAttribute() to determine the posted attribute's value
- ▶ Overriding remove() to delete associated child entities
- ▶ Overriding remove() to delete a parent entity in an association
- ▶ Using a method validator based on a view object accessor
- ▶ Using Groovy expressions to resolve validation error message tokens
- ▶ Using doDML() to enforce a detail record for a new master record

Introduction

Entity objects are the basic building blocks in the chain of business components. They represent a single row of data and they encapsulate the business model, data, rules, and persistence behavior. Usually, they map to database objects, most commonly to database tables, and views. Entity object definitions are stored in XML metadata files. These files are maintained automatically by JDeveloper and the ADF framework, and they should not be edited by hand. The default entity object implementation is provided by the ADF framework class `oracle.jbo.server.EntityImpl`. For large-scale projects you should create your own custom entity framework class, as demonstrated in the *Setting up BC base classes* recipe in *Chapter 1, Pre-requisites to Success: ADF Project Setup and Foundations*.

Likewise, it is not uncommon in large-scale projects to provide custom implementations for the entity object methods `doDML()`, `create()`, and `remove()`. The recipes in this chapter demonstrate, among other things, some of the custom functionality that can be implemented in these methods. Furthermore, other topics such as generic programming using custom properties and property sets, custom validators, entity associations, populating sequence attributes, and more, are covered throughout the chapter.

Using a custom property to populate a sequence attribute

In this recipe, we will go over a generic programming technique that you can use to assign database sequence values to specific entity object attributes. Generic functionality is achieved by using custom properties. Custom properties allow you to define custom metadata that can be accessed by the ADF business components at runtime.

Getting ready

We will add this generic functionality to the custom entity framework class. This class was created back in the *Setting up BC base classes* recipe in *Chapter 1, Pre-requisites to Success: ADF Project Setup and Foundations*. The custom framework classes in this case reside in the `SharedComponets` workspace. This workspace was created in the recipe *Breaking up the application in multiple workspaces, Chapter 1, Pre-requisites to Success: ADF Project Setup and Foundations*. You will need to create a database connection to the `HR` schema, if you are planning to run the recipe's test case. You can do this either by creating the database connection in the **Resource Palette** and dragging-and-dropping it to **Application Resources | Connections**, or by creating it directly in **Application Resources | Connections**.

How to do it...

1. Start by opening the `SharedComponets` workspace in JDeveloper. If needed, follow the steps in the referenced recipe to create it.
2. Locate the custom entity framework class in the `SharedBC` project and open it in the editor.
3. Click on the **Override Methods...** icon on the toolbar (the green left arrow) to bring up the **Override Methods** dialog.
4. From the list of methods that are presented, select the **create()** method and click **OK**. JDeveloper will insert a `create()` method in to the body of your custom entity class.

5. Add the following code to the `create()` method immediately after the call to `super.create()`:

```
// iterate all entity attributes
for (AttributeDef atrbDef :
    this.getEntityDef().getAttributeDefs()) {
    // check for a custom property called CREATESEQ_PROPERTY
    String sequenceName =
        (String)atrbDef.getProperty(CREATESEQ_PROPERTY);
    if (sequenceName != null) {
        // create the sequence based on the custom property sequence
        name
        SequenceImpl sequence = new SequenceImpl(sequenceName,
            this.getDBTransaction());
        // populate the attribute with the next sequence number
        this.populateAttributeAsChanged(atrbDef.getIndex(),
            sequence.getSequenceNumber());
    }
}
```

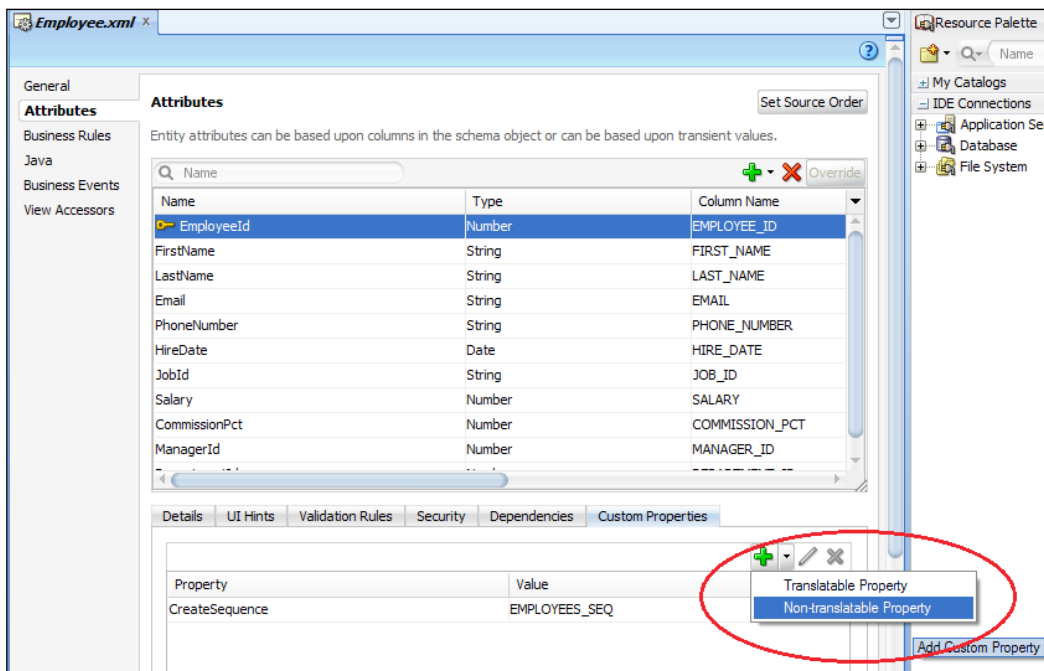
How it works...

In the previous code, we have overridden the `create()` method for the custom entity framework class. This method is called by the ADF framework each time a new entity object is constructed. We call `super.create()` to allow the framework processing, and then we retrieve the entity's attribute definitions by calling `getEntityDef().getAttributeDefs()`. We then iterate over them, calling `getProperty()` for each attribute definition. `getProperty()` accepts the name of a custom property defined for the specific attribute. In our case, the custom property is called `CreateSequence` and it is indicated by the constant definition `CREATESEQ_PROPERTY`, representing the name of the database sequence used to assign values to the particular attribute. Next, we instantiate a `SequenceImpl` object using the database sequence name retrieved from the custom property. Note that this does not create the database sequence, rather an `oracle.jbo.server.SequenceImpl` object representing a database sequence.


Finally, the attribute is populated with the value returned from the sequence—via the `getSequenceNumber()` call—by calling `populateAttributeAsChanged()`. This method will populate the attribute without marking the entity as changed. By calling `populateAttributeAsChanged()`, we will avoid any programmatic or declarative validations on the attribute while marking the attribute as changed, so that its value is posted during the entity object DML. Since all of the entity objects are derived from the custom entity framework class, all object creations will go through this `create()` implementation.

There's more...

So how do you use this technique to populate your sequence attributes? First you must deploy the `SharedComponents` workspace into an **ADF Library JAR** and add the library to the project where it will be used. Then, you must add the `CreateSequence` custom property to the specific attributes of your entity objects that need to be populated by a database sequence. To add a custom property to an entity object attribute, select the specific attribute in the entity **Attributes** tab and click on the arrow next to the **Add Custom Property** icon (the green plus sign) in the **Custom Properties** tab. From the context menu, select **Non-translatable Property**.



Click on the **Property** field and enter `CreateSequence`. For the **Value** enter the database sequence name that will be used to assign values to the specific attribute. For the **Employee** entity object example mentioned earlier, we will use the **EMPLOYEES_SEQ** database sequence to assign values to the **EmployeeId** attribute.


 Note that for testing purposes, we have created in the `HREmployees` workspace an `Employee` entity object and added the `CreateSequence` custom property to its `EmployeeId` attribute. To test the recipe, you can run the `EmployeeAppModule` application module.

See also

- ▶ *Breaking up the application in multiple workspaces, Chapter 1, Pre-requisites to Success: ADF Project Setup and Foundations*
- ▶ *Setting up BC base classes, Chapter 1, Pre-requisites to Success: ADF Project Setup and Foundations*
- ▶ *Overriding doDML() to populate an attribute with a gapless sequence, in this chapter*
- ▶ *Creating and applying property sets, in this chapter*

Overriding doDML() to populate an attribute with a gapless sequence

In this recipe, we will go over a generic programming technique that you can use to assign gapless database sequence values to entity object attributes. A gapless sequence will produce values with no gaps in between them. The difference between this technique and the one presented in the *Using a custom property to populate a sequence attribute* recipe, is that the sequence values are assigned during the transaction commit cycle instead of during component creation.

Getting ready

We will add this generic functionality to the custom entity framework class that we created in the *Setting up BC base classes* recipe in *Chapter 1, Pre-requisites to Success: ADF Project Setup and Foundations*. The custom framework classes in this case reside in the `SharedComponent.s` workspace. You will need access to the `HR` database schema to run the recipe's test case.

How to do it...

1. Start by opening the `SharedComponent.s` workspace in JDeveloper. If needed, follow the steps in the referenced recipe to create it.
2. Locate the custom entity framework class in the `SharedBC` project and open it in the editor.
3. Click on the **Override Methods...** icon on the toolbar (the green left arrow) to bring up the **Override Methods** dialog.
4. From the list of methods that are presented, select the **doDML()** method and click **OK**. JDeveloper will go ahead and insert a `doDML()` method into the body of your custom entity class.

5. Add the following code to the `doDML()` before the call to `super.doDML()`:

```
// check for insert operation
if (DML_INSERT == operation) {
    // iterate all entity attributes
    for (AttributeDef atrbDef :this.getEntityDef().
        getAttributeDefs()) {
        // check for a custom property called COMMITSEQ_PROPERTY
        String sequenceName=(String)atrDef.getProperty
            (COMMITSEQ_PROPERTY);
        if (sequenceName != null) {
            // create the sequence based on the custom property sequence
            name
            SequenceImpl sequence = new SequenceImpl(sequenceName,
                this.getDBTransaction());
            // populate the attribute with the next sequence number
            this.populateAttributeAsChanged(atrbDef.getIndex(),
                sequence.getSequenceNumber());
        }
    }
}
```

How it works...

If you examine the code presented in this recipe, you will see that it looks similar to the code presented in the *Using a custom property to populate a sequence attribute* recipe in this chapter. The difference is that this code executes during the transaction commit phase. During this phase, the ADF framework calls the entity's `doDML()` method. In our overridden `doDML()`, we first check for a `DML_INSERT` operation flag. This would be the case when inserting a new record into the database. We then iterate the entity's attribute definitions looking for a custom property identified by the constant `COMMITSEQ_PROPERTY`. Based on the property's value, we create a sequence object and get the next sequence value by calling `getSequenceNumber()`. Finally, we assign the sequence value to the specific attribute by calling `populateAttributeAsChanged()`. Assigning a sequence value during the commit phase does not allow the user to intervene. This will produce gapless sequence values. Of course to guarantee that there are no final gaps in the sequence values, deletion should not be allowed. That is, if rows are deleted, gaps in the sequence values will appear. Gaps will also appear in case of validation failures, if you do not subsequently rollback the transaction. Since all of the entity objects are derived from the custom entity framework class, all object commits will go through this `doDML()` implementation.

To use this technique, first you will need to re-deploy the shared components project. Then add the `CommitSequence` custom property as needed to the specific attributes of your entity objects. We explained how to do this in the *Using a custom property to populate a sequence attribute* recipe.

There's more...

`doDML()` is called by the ADF framework during a transaction commit operation. It is called for every entity object in the transaction's pending changes list. This is true even when entity **Update Batching** optimization is used. For an entity-based view object, this means that it will be called for every row in the row set that is in the pending changes list. The method accepts an operation flag; `DML_INSERT`, `DML_UPDATE`, or `DML_DELETE` to indicate an insert, update, or delete operation on the specific entity.

Data is posted to the database once `super.doDML()` is called, so any exceptions thrown before calling `super.doDML()` will result in no posted data. Once the data is posted to the database, queries or stored procedures that rely upon the posted data should be coded in the overridden application module's `beforeCommit()` method. This method is also available at the entity object level, where it is called by the framework for each entity in the transaction's pending changes list. Note that the framework calls `beforeCommit()` for each entity object in the *transaction pending changes list* prior to calling the application module `beforeCommit()`.

For additional information on `doDML()`, consult the sections *Methods You Typically Override in Your Custom EntityImpl Subclass* and *Transaction "Post" Processing (Record Cache)* in the *Fusion Developer's Guide for Oracle Application Development Framework*, which can be found at http://docs.oracle.com/cd/E24382_01/web.1112/e16182/toc.htm.



Note that for testing purposes, we have created a `Department` entity object in the `HRDepartments` workspace and added the `CommitSequence` custom property to its `DepartmentId` attribute. The value of the `CommitSequence` property was set to `DEPARTMENTS_SEQ`, the database sequence that is used to assign values to the `DepartmentId` attribute. To test the recipe, run the `DepartmentAppModule` application module on the **ADF Model Tester**.

See also

- ▶ *Breaking up the application in multiple workspaces, Chapter 1, Pre-requisites to Success: ADF Project Setup and Foundations*
- ▶ *Setting up BC base classes, Chapter 1, Pre-requisites to Success: ADF Project Setup and Foundations*
- ▶ *Using a custom property to populate a sequence attribute, in this chapter*
- ▶ *Creating and applying property sets, in this chapter*

Creating and applying property sets

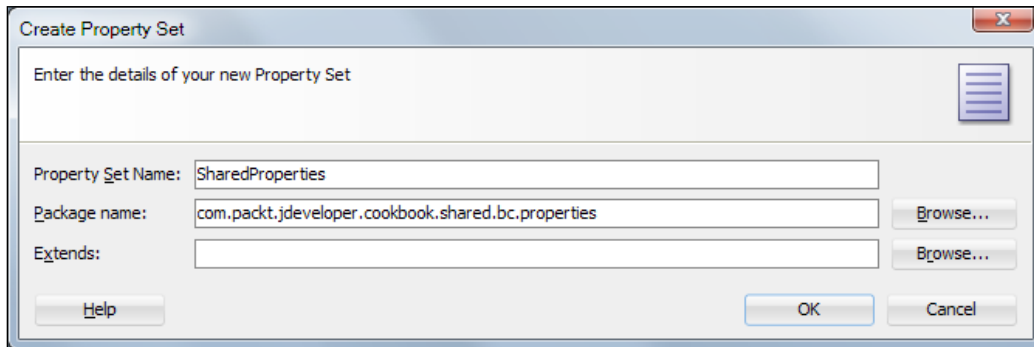
In the *Using a custom property to populate a sequence attribute* and *Overriding doDML() to populate an attribute with a gapless sequence* recipes of this chapter, we introduced custom properties for generic ADF business component programming. In this recipe, we will present a technique to organize your custom properties in reusable **property sets**. By organizing application-wide properties in a property set and exporting them as part of an ADF Library JAR, you can then reference them from any other ADF-BC project. This in turn will allow you to centralize the custom properties used throughout your ADF application in a single property set.

getting ready

We will create a property set in the `SharedComponets` workspace. I suggest that you go over the *Using a custom property to populate a sequence attribute* and *Overriding doDML() to populate an attribute with a gapless sequence* recipes in this chapter before continuing with this recipe. To run the recipe's test cases, you will need access to the `HR` schema in the database.

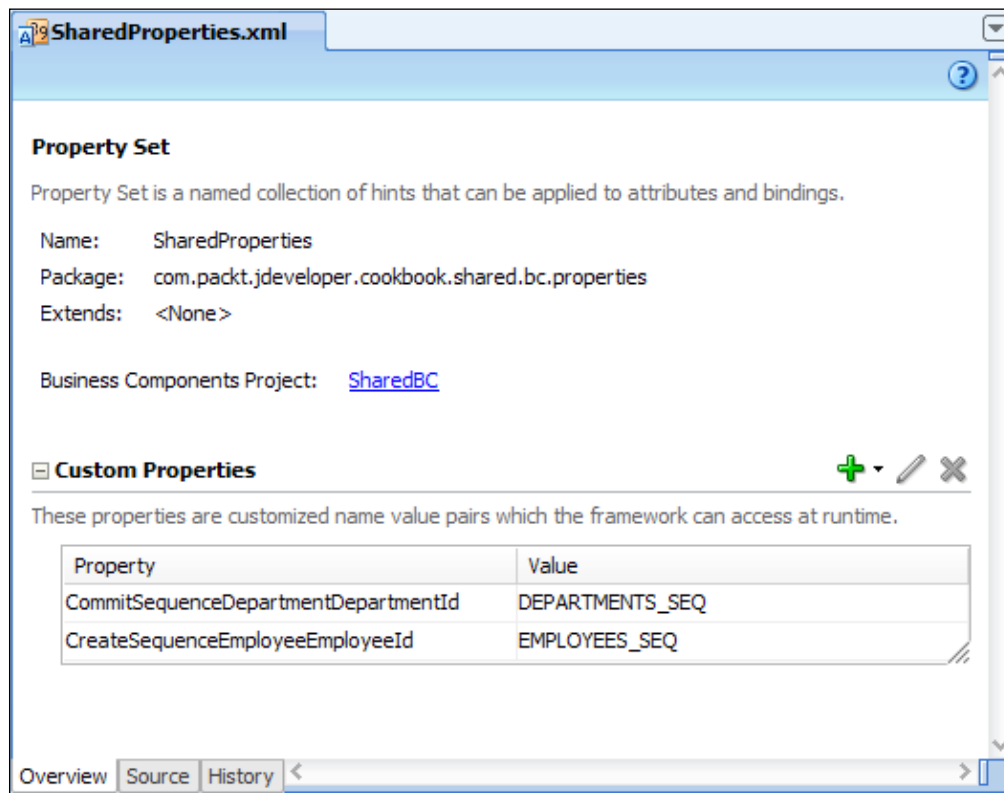
How to do it...

1. Start by opening the `SharedComponets` workspace. If needed, follow the steps in the referenced recipe to create it.
2. Right-click on the `SharedBC` project and select **New...**
3. On the **New Gallery** dialog select **ADF Business Components** under the **Business Tier** node and **Property Set** from the **Items** on the right.
4. Click **OK** to proceed. This will open the **Create Property Set** dialog.



5. Enter the property set name and package in the appropriate fields. For this recipe, we will call it `SharedProperties` and use the `com.packt.jdeveloper.cookbook.shared.bc.properties` package. Click **OK** to continue.

6. JDeveloper will create and open the `SharedProperties` property set.
7. To add a custom property to the property set, click on the **Add Custom Property** button (the green plus sign icon).
8. Go ahead and add two non-translatable properties called **`CommitSequenceDepartmentDepartmentId`** and **`CreateSequenceEmployeeEmployeeId`**. Set their values to **`DEPARTMENTS_SEQ`** and **`EMPLOYEES_SEQ`** respectively. Your property set should look similar to the following screenshot:



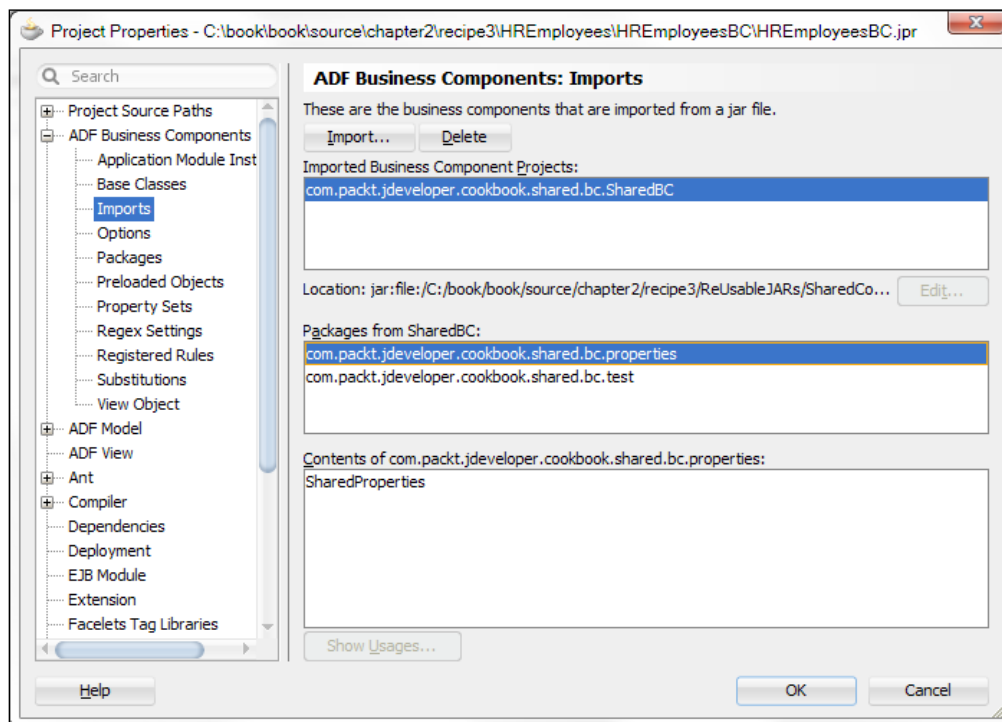
9. Next you need to change the `create()` method in the custom entity framework class so that the custom property is now similar to the following block of code:

```
// construct the custom property name from the entity name and
attribute
String propertyName = CREATESEQ_PROPERTY +
    getEntityDef().getName() + atrbDef.getName();
// check for a custom property called CREATESEQ_PROPERTY
String sequenceName = (String) atrbDef.getProperty(propertyName);
```

- Similarly change the `doDML()` method in the custom entity framework class so that the custom property is also constructed, as shown in the following block of code:

```
// construct the custom property name from the entity name and
// attribute
String propertyName = COMMITSEQ_PROPERTY +
    getEntityDef().getName() + atrbDef.getName();
// check for a custom property called COMMITSEQ_PROPERTY
String sequenceName = (String) atrbDef.getProperty(propertyName);
```

- Redeploy the `SharedComponents` workspace into an ADF Library JAR.
- Open the `HREmployees` workspace and double-click on the `HREmployeesBC` business components project to bring up the **Project Properties** dialog.
- Select **Imports** under the **ADF Business Components** node and click on the **Import...** button on the right.
- On the **Import Business Components XML File** dialog browse for the shared components ADF Library JAR file in the `ReusableJARs` directory. Select it and click **Open**.
- You should see the imported `SharedBC` project under the **Imported Business Component Projects** along with the imported packages and package contents. Click **OK** to continue with importing the business components.



16. Double-click on the **Employee** entity object and go to the **Attributes** tab.
17. Click on the **Details** tab, and from the **Property Set** choice list select the imported property set.
18. Repeat steps 12-17 for the `HRDepartments` workspace and apply the property set to the `DepartmentId` attribute of the `Department` entity object.

How it works...

Property sets are a way to gather all of your custom properties together into logical collections. Instead of applying each custom property separately to a business components object or to any of its attributes, custom properties defined in these collections can be applied at once on them. Property sets can be applied to entity objects and their attributes, view objects and their attributes, and application modules. You access custom properties programmatically as indicated earlier, by calling `AttributeDef.getProperty()` for properties applied to attributes, `EntityDefImpl.getProperty()` for properties applied to entity objects, `ViewDefImpl.getProperty()` for properties applied to view objects, and so on.

How you organize your custom properties into property sets is up to you. In this recipe, for example, we use a single property set called `SharedProperties`, which we define in the shared components ADF library. In this way, we kept all custom properties used by the application in a single container. For this to work, we had to devise a way to differentiate among them. The algorithm that we used was to combine the property name with the business components object name and the attribute name that the property applies to. So we have properties called `CommitSequenceDepartmentDepartmentId` and `CreateSequenceEmployeeEmployeeId`.

Finally, we import the property set from the `SharedComponets` workspace into the relevant business components projects using the **Import Business Components** facility of the business components **Project Properties** dialog.

There's more...

To test the recipe, you can run the `EmployeeAppModule` and `DepartmentAppModule` application modules in the `HREmployees` and `HRDepartments` workspaces respectively.



Note that you can override any of the properties defined in a property set by explicitly adding the same property to the business component object or to any of its attributes.

Also note that property sets can be applied onto entity objects, view objects, and application modules by clicking on the **Edit property set selection** button (the pen icon) on the business component object definition **General** tab. On the same tab, you can add custom properties to the business component object by clicking on the **Add Custom Property** button (the green plus sign icon).

See also

- ▶ *Breaking up the application in multiple workspaces, Chapter 1, Pre-requisites to Success: ADF Project Setup and Foundations*
- ▶ *Setting up BC base classes, Chapter 1, Pre-requisites to Success: ADF Project Setup and Foundations*
- ▶ *Using a custom property to populate a sequence attribute, in this chapter*
- ▶ *Overriding doDML() to populate an attribute with a gapless sequence, in this chapter*

Using `getPostedAttribute()` to determine the posted attribute's value

There are times when you need to get the original database value of an entity object attribute, such as when you want to compare the attribute's current value to the original database value. In this recipe, we will illustrate how to do this by utilizing the `getPostedAttribute()` method.

Getting ready

We will be working on the `SharedComponets` workspace. We will add a helper method to the custom entity framework class.

How to do it...

1. Start by opening the `SharedComponets` workspace. If needed, follow the steps in the referenced recipe to create it.
2. Locate the custom entity framework class and open it into the source editor.

3. Add the following code to the custom entity framework class:

```
/**
 * Check if attribute's value differs from its posted value
 * @param attrIdx the attribute index
 * @return
 */
public boolean isAttrValueChanged(int attrIdx) {
    // get the attribute's posted value
    Object postedValue = getPostedAttribute(attrIdx);
    // get the attribute's current value
    Object newValue = getAttributeInternal(attrIdx);
    // return true if attribute value differs from its posted value
    return isAttributeChanged(attrIdx) &&
        ((postedValue == null && newValue != null) ||
         (postedValue != null && newValue == null) ||
         (postedValue != null && newValue != null &&
          !newValue.equals(postedValue)));
}
```

How it works...

We added a helper method called `isAttrValueChanged()` to the our custom entity framework class. This method accepts the attribute's index. The attribute index is generated and maintained by JDeveloper itself. The method first calls `getPostedAttribute()` specifying the attribute index to retrieve the attribute value that was posted to the database. This is the attribute's database value. Then it calls `getAttributeInternal()` using the same attribute index to determine the current attribute value. The two values are then compared. The method `isAttributeChanged()` returns `true` if the attribute value was changed in the current transaction.

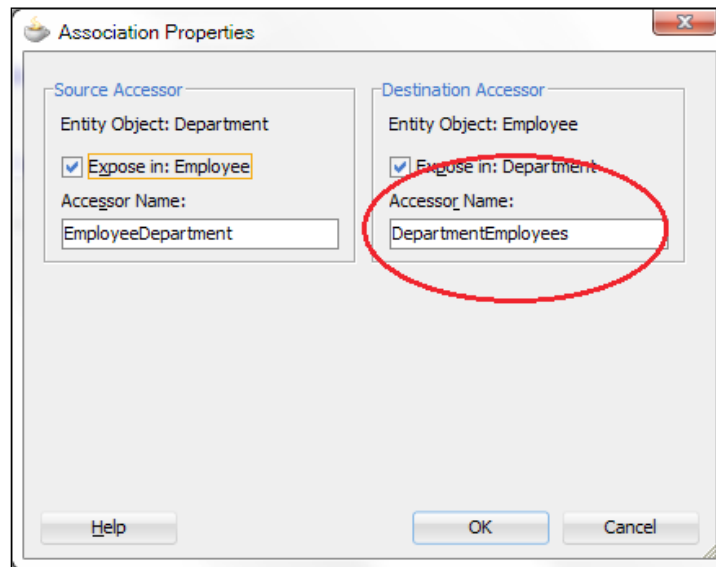
The following is an example of calling `isAttrValueChanged()` from an entity implementation class to determine whether the current value of the employee's last name differs from the value that was posted to the database:

```
super.isAttrValueChanged(this.LASTNAME);
```

See also

- ▶ *Breaking up the application in multiple workspaces, Chapter 1, Pre-requisites to Success: ADF Project Setup and Foundations*
- ▶ *Setting up BC base classes, Chapter 1, Pre-requisites to Success: ADF Project Setup and Foundations*

4. Double-click on the **EmpDeptFkAssoc** association on the **Application Navigator** to open the association definition, then click on the **Relationship** tab.
5. Click on the **Edit accessors** button (the pen icon) in the **Accessors** section to bring up the **Association Properties** dialog.
6. Change the **Accessor Name** in the **Destination Accessor** section to **DepartmentEmployees** and click **OK** to continue.



7. Double-click on the **Department** entity object in the **Application Navigator** to open its definition and go to the **Java** tab.
8. Click on the **Edit Java Options** button (the pen icon on the top right of the tab) to bring up the **Select Java Options** dialog.
9. On the **Select Java Options** dialog, select **Generate Entity Object Class**.
10. Ensure that both the **Accessors** and **Remove Method** checkboxes are selected. Click **OK** to continue.
11. Repeat steps 7-10 to create a Java implementation class for the `Employee` entity object. You do not have to click on the **Remove Method** checkbox in this case.
12. Open the `DepartmentImpl` Java implementation class for the `Department` entity object in the JDeveloper Java editor and locate the `remove()` method.

13. Add the following code before the call to `super.remove()`:

```
// get the department employees accessor
RowIterator departmentEmployees = this.getDepartmentEmployees();
// iterate over all department employees
while (departmentEmployees.hasNext()) {
    // get the department employee
    EmployeeImpl departmentEmployee =
        (EmployeeImpl) departmentEmployees.next();
    // remove employee
    departmentEmployee.remove();
}
```

How it works...

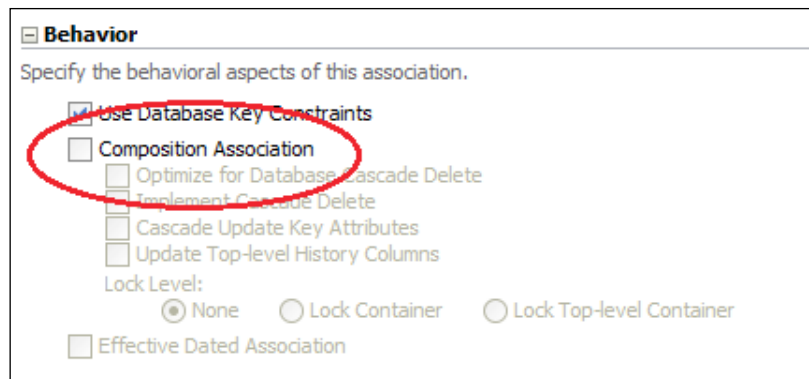
During the creation of the `Department` and `Employee` entity objects, `JDeveloper` automatically creates the entity associations based on the foreign key constraints that exist among the `DEPARTMENTS` and `EMPLOYEES` database tables. The specific association that relates a department to its employees was automatically created and it was called `EmpDeptFkAssoc`.

`JDeveloper` exposes the association to both the source and destination entity objects via accessors. In step 6, we changed the accessor name to make it more meaningful. We called the the association accessor that returns the department employees `DepartmentEmployees`. Using Java, this accessor is available in the `DepartmentImpl` class by calling `getDepartmentEmployees()`. This method returns an `oracle.jbo.RowIterator` object that can be iterated over.

Now, let's take a closer look at the code added to the `remove()` method. This method is called by the ADF framework each time we delete a record. In it, first we access the current department's employees by calling `getDepartmentEmployees()`. Then we iterate over the department employees, by calling `hasNext()` on the employees `RowIterator`. Then for each employee, we get the `Employee` entity object by calling `next()`, and call `remove()` on it to delete it. The call to `super.remove()` finally deletes the `Department` entity itself. The net result is to delete all employees associated with the specific department before deleting the department itself.

There's more...

A specific type of association called composition association can be enabled in those cases where an object composition behavior is observed, that is, where the child entity cannot exist on its own without the associated "parent" entity. In these cases, there are special provisions by the ADF framework and `JDeveloper` itself to fine-tune the delete behavior of child entities when the parent entity is removed. These options are available in the association editor **Relationship** tab, under the **Behavior** section.



Behavior

Specify the behavioral aspects of this association.

Use Database Key Constraints

Composition Association

Optimize for Database Cascade Delete

Implement Cascade Delete

Cascade Update Key Attributes

Update Top-level History Columns

Lock Level:

None Lock Container Lock Top-level Container

Effective Dated Association

Once you indicate a **Composition Association** for the association, two options are presented relating to cascading deletion:

- ▶ **Optimize for Database Cascade Delete:** This option prevents the framework from issuing a `DELETE` DML statement for each composed entity object destination row. You do this if `ON DELETE CASCADE` is implemented in the database.
- ▶ **Implement Cascade Delete:** This option implements the cascade delete in the middle layer, that is if the source composing entity object contains any composed children, its deletion is prevented.

This recipe shows how to remove children entity objects for which composition association is not enabled. This may be the case when a requirement exists to allow in some cases composed children entities to exist without associated composing parent entities. For example, when a new employee is not yet assigned to a particular department.

Overriding `remove()` to delete a parent entity in an association

In this recipe, we will present a technique that you can use in cases that you want to delete the parent entity in an association when the last child entity is deleted. An example of such a case would be to delete a department when the last department employee is deleted.

Getting ready

You will need access to the `HR` schema in your database.

How to do it...

1. Start by creating a new **Fusion Web Application (ADF)** workspace called `HRComponents`.
2. Create a database connection for the `HR` schema in the **Application Resource** section of the **Application Navigator**.
3. Use the **Business Components from Tables** selection on the **New Gallery** dialog to create `Business Components` objects for the `DEPARTMENTS` and `EMPLOYEES` tables.
4. Double-click on the **EmpDeptFkAssoc** association on the **Application Navigator** to bring up the Association editor, then click on the **Relationship** tab.
5. Click on the **Edit accessors** button (the pen icon) in the **Accessors** section to bring up the **Association Properties** dialog.
6. Change the **Accessor Name** in the **Source Accessor** section to **EmployeeDepartment** and click **OK** to continue.
7. Generate custom Java implementation classes for both the `Employee` and `Department` entity objects.
8. Open the `EmployeeImpl` custom Java implementation class for the `Employee` entity object and locate the `remove()` method.
9. Replace the call to `super.remove()` with the following code:

```
// get the associated department
DepartmentImpl department = this.getEmployeeDepartment();
// get number of employees in the department
int numberOfEmployees =
    department.getDepartmentEmployees().getRowCount();
// check whether last employee in the department
if (numberOfEmployees == 1) {
    // delete the last employee
    super.remove();
    // delete the department as well
    department.remove();
}
else {
    // just delete the employee
    super.remove();
}
```

How it works...

If you followed the *Overriding remove() to delete associated children entities* recipe in this chapter, then steps 1 through 8 should look familiar. These are the basic steps to create the `HRComponents` workspace, along with the business components associated with the `EMPLOYEES` and `DEPARTMENTS` tables in the HR schema. These steps also create custom Java implementation classes for the `Employee` and `Department` entity objects and setup the `EmpDeptFkAssoc` association.

The code in `remove()` first gets the `Department` entity row by calling the accessor `getEmployeeDepartment()` method. Remember, this was the name of accessor—`EmployeeDepartment`—that we setup in step 6. `getEmployeeDepartment()` returns the custom `DepartmentImpl` that we setup in step 7. In order to determine the number of employees in the associated `Department`, we first get the `Employee RowIterator` by calling `getDepartmentEmployees()` on it, and then `getRowCount()` on the `RowIterator`. All that is done in the following statement:

```
int numberOfEmployees =
    department.getDepartmentEmployees().getRowCount();
```

Remember that we setup the name of the `DepartmentEmployees` accessor in step 6. Next, we checked for the number of employees in the associated department, and if there was only one employee—the one we are about to delete—we first deleted it by calling `super.remove()`. Then we deleted the department itself by calling `department.remove()`. If more than one employee was found for the specific department, we just delete the employee by calling `super.remove()`. This was done in the `else` part of the `if` statement.

There's more...

Note the implications of using `getRowCount()` versus `getEstimatedRowCount()` in your code when dealing with large result sets: `getRowCount()` will perform a database count query each time it is called to return the exact number of rows in the view object. On the other hand, `getEstimatedRowCount()` executes a database count query only once to fetch the view object row count to the middle layer. Then, it fetches the row count from the middle layer. The row count in the middle layer is adjusted as view object rows are added or deleted. This may not produce an accurate row count when multiple user sessions are manipulating the same view object at the same time. For more information on this topic, consult the section *How to Count the Number of Rows in a Row Set* in the *Fusion Developer's Guide for Oracle Application Development Framework*.

See also

- ▶ *Overriding remove() to delete associated children entities*, in this chapter

Using a method validator based on a view object accessor

In this recipe, we will show how to validate an entity object against a **view accessor** using a custom entity method validator. The use case that we will cover—based on the HR schema—will not allow the user to enter more than a specified number of employees per department.

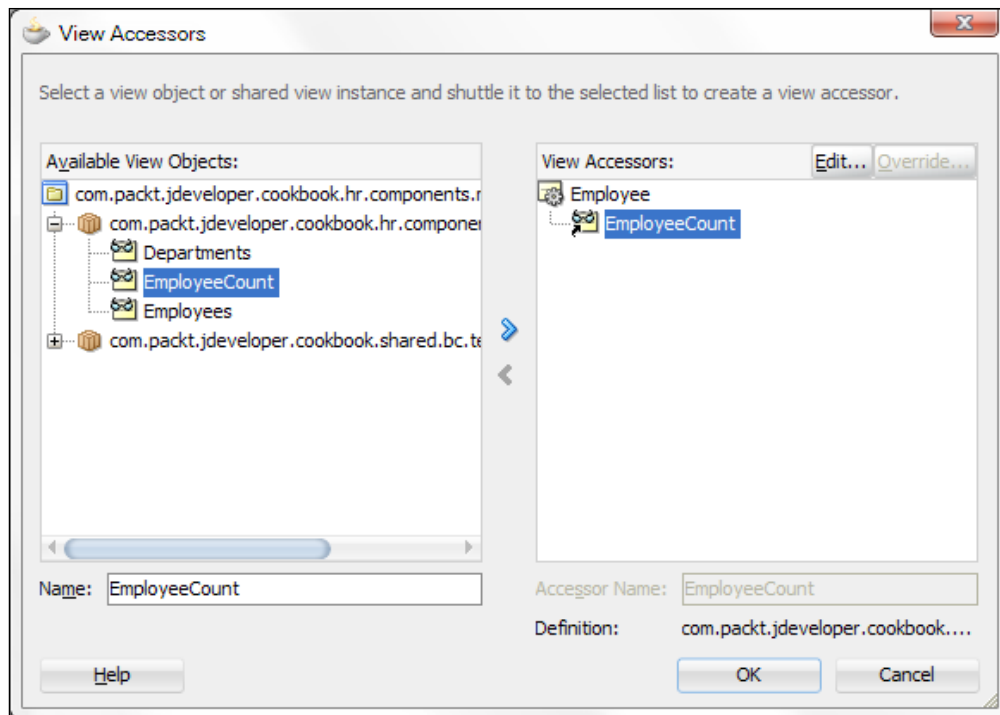
Getting ready

We will be using the `HRComponents` workspace that we created in the previous recipes in this chapter so that we don't repeat these steps again. You will need access to the HR database schema.

How to do it...

1. Right-click on the `com.packt.jdeveloper.cookbook.hr.components.model.view` package of the `HRComponentsBC` business components project of the `HRComponents` workspace, and select **New View Object...**
2. Use the **Create View Object** wizard to create a **SQL query** view object called **EmployeeCount** based on the following query:

```
SELECT COUNT(*) AS EMPLOYEE_COUNT FROM EMPLOYEES WHERE DEPARTMENT_ID = :DepartmentId
```
3. While on the **Create View Object** wizard, also do the following:
 - ❑ Create a **Bind Variable** called **DepartmentId** of type **Number**
 - ❑ On the **Attribute Settings** page, ensure that you select **Key Attribute** for the **EmployeeCount** attribute
 - ❑ On the **Java** page make sure that both the **Generate View Row Class** and **Include accessors** checkboxes are checked
 - ❑ Do not add the view object to an application module
4. Now, double-click on the **Employee** entity object to open its definition and go to the **View Accessors** page.
5. Click on the **Create new view accessors** button (the green plus sign icon) to bring up the **View Accessors** dialog.
6. On the **View Accessors** dialog locate the **EmployeeCount** view object and click the **Add instance** button—the blue right arrow button. Click **OK** to dismiss the dialog.



7. On the entity object definition **Business Rules** tab, select the **Employee** entity and click on the **Create new validator** button (the green plus sign icon).
8. On the **Add Validation Rule** dialog, select **Method** for the **Rule Type** and enter `validateDepartmentEmployeeCount` for the **Method Name**.
9. Click on the **Failure Handling** tab and in the **Message Text** enter the message `Department has reached maximum employee limit.` Click **OK**.
10. Open the `EmployeeImpl` custom implementation Java class, locate the `validateDepartmentEmployeeCount()` method and add the following code to it before the `return true` statement:

```
// get the EmployeeCount view accessor
RowSet employeeCount = this.getEmployeeCount();
// setup the DepartmentId bind variable
employeeCount.setNamedWhereClauseParam("DepartmentId",
    this.getDepartmentId());
// run the View Object query
employeeCount.executeQuery();
// check results
if (employeeCount.hasNext()) {
    // get the EmployeeCount row
```

```
EmployeeCountRowImpl employeeCountRow =
    (EmployeeCountRowImpl) employeeCount.next();
// get the department employee count
Number departmentEmployees =
    employeeCountRow.getEmployeeCount();
if (departmentEmployees.compareTo(MAX_DEPARTMENT_EMPLOYEES) > 0) {
    return false;
}
}
```

How it works...

We have created a separate query-based view object called `EmployeeCount` for validation purposes. If you look closely at the `EmployeeCount` query, you will see that it determines the number of employees in a department. Which department is determined by the bind variable `DepartmentId` used in the `WHERE` clause of the query.

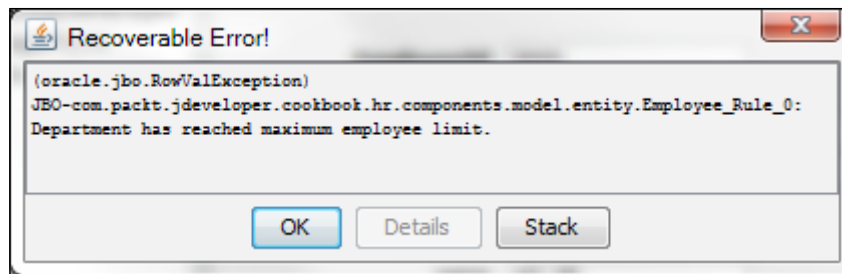
We then add the `EmployeeCount` view object as a view accessor to the `Employee` object. We call the accessor instance `EmployeeCount` as well. Once you have generated a custom Java implementation class for the `Employee` entity object, the `EmployeeCount` view accessor is available by calling `getEmployeeCount()`.

We proceed by adding a method validator to the entity object. We call the method to use for the validator `validateDepartmentEmployeeCount`. `JDeveloper` created this method for us in the entity custom implementation Java class.

The code that we add to the `validateDepartmentEmployeeCount()` method first gets the `EmployeeCount` accessor, and calls `setNamedWhereClauseParam()` on it to set the value of the `DepartmentId` bind variable to the value of the department identifier from the current `Employee`. This value is accessible via the `getDepartmentId()` method. We then execute the `EmployeeCount` view object query by calling its `executeQuery()` method. We check for the results of the query by calling `hasNext()` on the view object. If the query yields results, we get the next result row by calling `next()`. We have casted the `oracle.job.Row` returned by `next()` to an `EmployeeCountRowImpl` so we can directly call its `getEmployeeCount()` accessor. This returns the number of employees for the specific department. We then compare it to a predefined maximum number of employees per department identified by the constant `MAX_DEPARTMENT_EMPLOYEES`.

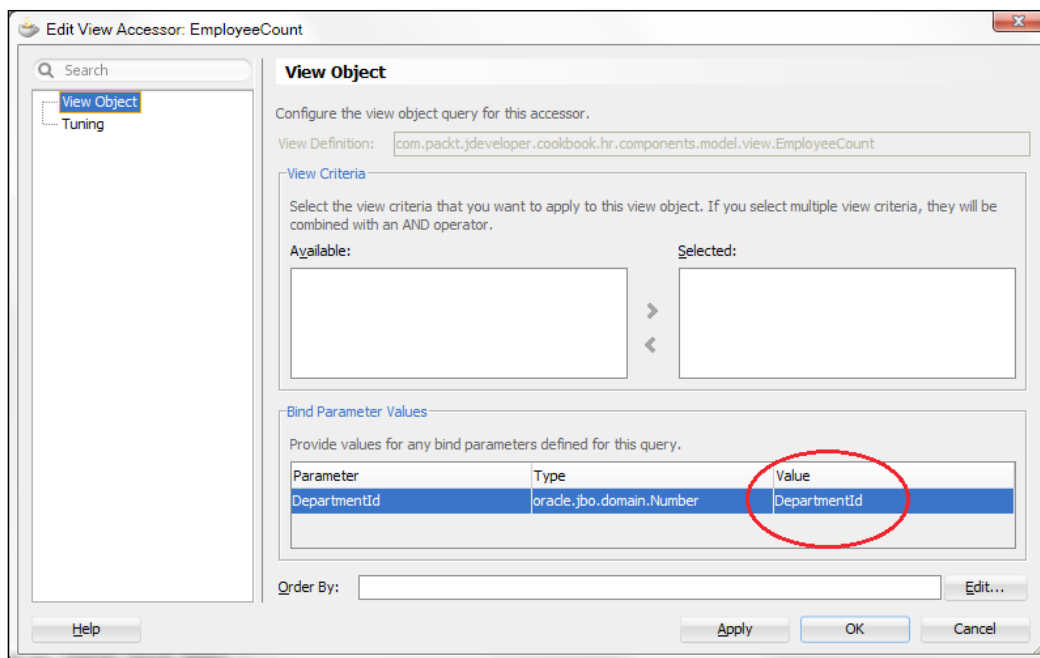
The method validator returns a `false` to indicate that the validation will fail. Otherwise it returns `true`.

Observe what happens when you run the application module with the ADF Model Tester. When you try to add a new employee to a department that has more than a predefined number of employees (identified by the constant `MAX_DEPARTMENT_EMPLOYEES`), a validation message is raised. This is the message that we defined for our method validator.



There's more...

Note that in the previous code we called `setNamedWhereClauseParam()` on the `EmployeeCount` view object to set the value of the `DepartmentId` bind variable to the current employee's department ID. This could have been done declaratively as well using the **Edit View Accessor** dialog, which is available on the **View Accessors** page of the `Employee` entity definition page by clicking on the **Edit selected View Accessor** button (the pen icon). On the **Edit View Accessor** dialog, locate the `DepartmentId` bind variable in the **Bind Parameter Values** section, and on the **Value** field enter `DepartmentId`. This will set the value of the `DepartmentId` bind variable to the value of the `DepartmentId` attribute of the `Employee` entity object.



See also

- ▶ *Overriding remove() to delete associated children entities*, in this chapter

Using Groovy expressions to resolve validation error message tokens

In this recipe, we will expand on the *Using a custom validator based on a View Object accessor* recipe to demonstrate how to use validation message parameter values based on Groovy expressions. Moreover, we will show how to retrieve the parameter values from a specific parameter bundle.

Groovy is a dynamic language that runs inside the Java Virtual Machine. In the context of the ADF Business Components framework, it can be used to provide declarative expressions that are interpreted at runtime. Groovy expressions can be used in validation rules, validation messages, and parameters, attribute initializations, bind variable initializations, and more.

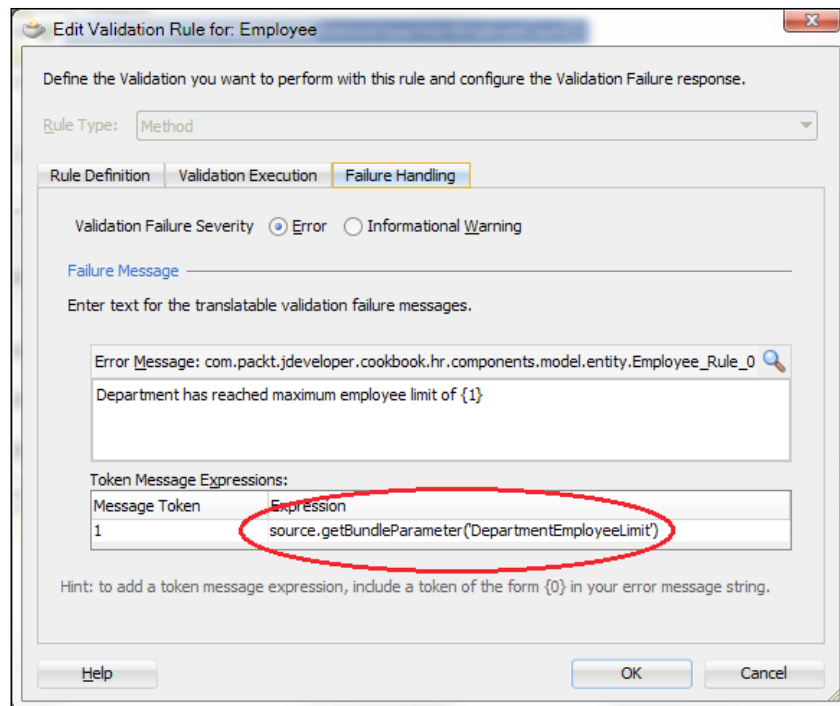
Getting ready

This recipe builds on the *Using a custom validator based on a View Object accessor* recipe. It also relies on the recipes *Breaking up the application in multiple workspaces* and *Setting up BC base classes* presented in *Chapter 1, Pre-requisites to Success: ADF Project Setup and Foundations*.

How to do it...

1. In the **Application Navigator** double-click on the **Employee** entity object definition and go to its **Business Rules** tab.
2. Double-click on the **validateDepartmentEmployeeCount Method Validator** to bring up the **Edit Validation Rule** dialog and go to the **Failure Handling** tab.
3. Change the **Error Message** to *Department has reached maximum employee limit of {1}*.
4. For the **Message Token 1 Expression** in the **Token Message Expressions** section, enter the following expression:

```
source.getBundleParameter('DepartmentEmployeeLimit')
```



5. Now, open the SharedComponents workspace and locate the entity framework extension class `ExtEntityImpl`. Add the following `getBundleParameter()` method to it:


```
public String getBundleParameter(String parameterKey) {
    // use BundleUtils to load the parameter
    return BundleUtils.loadParameter(parameterKey);
}
```
6. Locate the `BundleUtils` helper class in the `com.packt.jdeveloper.cookbook.shared.bc.exceptions.messages` package and add the following `loadParameter()` method:


```
public static String loadParameter(final String parameterKey) {
    // get access to the error message parameters bundle
    final ResourceBundle parametersBundle =
        ResourceBundle.getBundle(PARAMETERS_BUNDLE,
            Locale.getDefault());
    // get and return the the parameter value
    return parametersBundle.getString(PARAMETER_PREFIX +
        parameterKey);
}
```

7. Finally, locate the `ErrorParams.properties` property file and add the following text to it:

```
parameter.DepartmentEmployeeLimit=2
```

How it works...

For this recipe, first we added a parameter to the method validator message. The parameter is indicated by adding parameter placeholders to the message using braces `{}`. The parameter name is indicated by the value within the braces. In our case, we defined a parameter called `1` by entering `{1}`. We then had to supply the parameter value. Instead of hardcoding the parameter value, we used the following Groovy expression:

```
source.getBundleParameter('DepartmentEmployeeLimit').
```

The `source` prefix allows us to reference an entity object method from the validator. In this case, the method is called `getBundleParameter()`. This method accepts a parameter key which is used to load the actual parameter value from the parameters bundle. In this case, we have used the `DepartmentEmployeeLimit` parameter key.

Then we implemented the `getBundleParameter()` method. We implemented this method in the base entity custom framework class so that it is available to all entity objects. If you look at the code in `getBundleParameter()`, you will see that it loads and returns the parameter value using the helper `BundleUtils.loadParameter()`.



We introduced the helper class `BundleUtils` while we worked on the *Using a generic backing bean actions framework* recipe in *Chapter 1, Pre-requisites to Success: ADF Project Setup and Foundations*.

The `BundleUtils.loadParameter()` method pre-pends the parameter with the prefix `parameter`.

Finally, we defined the `parameter.DepartmentEmployeeLimit` parameter in the `ErrorParams.properties` parameters bundle. For further information on this bundle, refer to the *Using a custom exception class* recipe in *Chapter 1, Pre-requisites to Success: ADF Project Setup and Foundations*. When the validation is raised at runtime, the message parameter placeholder `{1}`, which was originally defined in the message, will be substituted with the actual parameter value (in this case, the number 2).

See also

- ▶ *Breaking up the application in multiple workspaces, Chapter 1, Pre-requisites to Success: ADF Project Setup and Foundations*
- ▶ *Setting up BC base classes, Chapter 1, Pre-requisites to Success: ADF Project Setup and Foundations*

- ▶ *Using a custom exception class, Chapter 1, Pre-requisites to Success: ADF Project Setup and Foundations*
- ▶ *Using a generic backing bean actions framework, Chapter 1, Pre-requisites to Success: ADF Project Setup and Foundations*

Using doDML() to enforce a detail record for a new master record

In this recipe, we will consider a simple technique that we can use to enforce having detailed records when inserting a new master record in an entity association relationship. The use case demonstrates how to enforce creating at least one employee at the time when a new department is created.

Getting ready

We will use the HR database schema and the HRComponents workspace that we have created in previous recipes in this chapter.

How to do it...

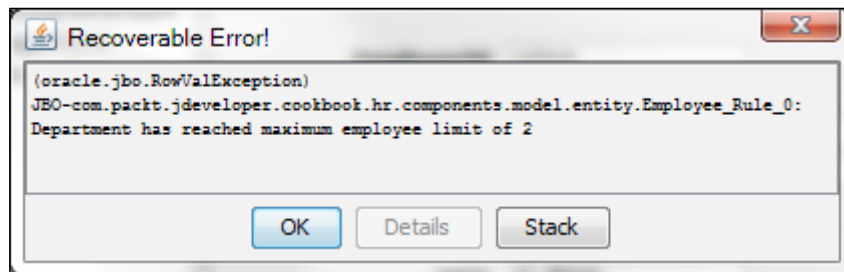
1. Open the `DepartmentImpl` custom entity implementation class and override the `doDML()` method using the **Override Methods** dialog.
2. Add the following code to the `doDML()` method before the call to `super.doDML()`:


```
// check for insert
if (DML_INSERT == operation) {
    // get the department employees accessor
    RowIterator departmentEmployees = this.getDepartmentEmployees();
    // check for any employees
    if (!departmentEmployees.hasNext()) {
        // avoid inserting the department if there are no employees
        for it
        throw new ExtJboException("00006");
    }
}
```


How it works...

In the overridden `doDML()`, we only check for insert operations. This is indicated by comparing the DML operation flag which is passed as a parameter to `doDML()` to the `DML_INSERT` flag. Then we get the department employees from the `DepartmentEmployees` accessor by calling `getDepartmentEmployees()`. The `DepartmentEmployees` accessor was set up during the creation of the `HRComponents` workspace earlier in this chapter. We check whether the `RowIterator` returned has any rows by calling `hasNext()` on it. If this is not the case, that is, there are no employees associated with the specific department that we are about to insert, we alert the user by throwing an `ExtJboException` exception. The `ExtJboException` exception is part of the `SharedComponets` workspace and it was developed in the *Using a custom exception class* recipe back in *Chapter 1, Pre-requisites to Success: ADF Project Setup and Foundations*.

When testing the application module with the ADF Model Tester, we get the following error message when we try to insert a new department without any associated employees:



 Note that in case that an exception is thrown during DML, which could result in partial data being posted to the database.

See also

- ▶ *Using a custom exception class, Chapter 1, Pre-requisites to Success: ADF Project Setup and Foundations*
- ▶ *Overriding `remove()` to delete associated children entities, Chapter 1, Pre-requisites to Success: ADF Project Setup and Foundations*

3

A Different Point of View: View Object Techniques

In this chapter, we will cover:

- ▶ Iterating a view object using a secondary rowset iterator
- ▶ Setting default values for view row attributes
- ▶ Controlling the updatability of view object attributes programmatically
- ▶ Setting the Queryable property of a view object attribute programmatically
- ▶ Using a transient attribute to indicate a new view object row
- ▶ Conditionally inserting new rows at the end of a rowset
- ▶ Using `findAndSetCurrentRowByKey()` to set the view object currency
- ▶ Restoring the current row after a transaction rollback
- ▶ Dynamically changing the WHERE clause of the view object query
- ▶ Removing a row from a rowset without deleting it from the database

Introduction

View objects are an essential part of the ADF business components. They work in conjunction with entity objects, making entity-based view objects, to support querying the database, retrieving data from the database, and building rowsets of data. The underlying entities enable an updatable data model that supports the addition, deletion, and modification of data. They also support the enforcement of business rules and the permanent storage of the data to the database.

In cases where an updatable data model is not required, the framework supports a read-only view object, one that is not based on entity objects but on a SQL query supplied by the developer. Read-only view objects should be used in cases where `UNION` and `GROUP BY` clauses appear in the view object queries. In other cases, even though an updatable data model is not required, the recommended practice is to base the view objects on entity objects and allow the JDeveloper framework-supporting wizards to build the SQL query automatically instead.

This chapter presents several techniques covering a wide area of expertise related to view objects.

Iterating a view object using a secondary rowset iterator

There are times when you need to iterate through a view object rowset programmatically. In this recipe, we will see how to do this using a secondary rowset iterator. We will iterate over the `Employees` rowset and increase the employee's commission by a certain percentage for each employee that belongs to the Sales department.

Getting ready

This recipe was developed using the `HRComponents` workspace, which was created in the *Overriding `remove()` to delete associated children entities* recipe in *Chapter 2, Dealing with Basics: Entity Objects*. The `HRComponents` workspace requires a database connection to the HR schema.

How to do it...

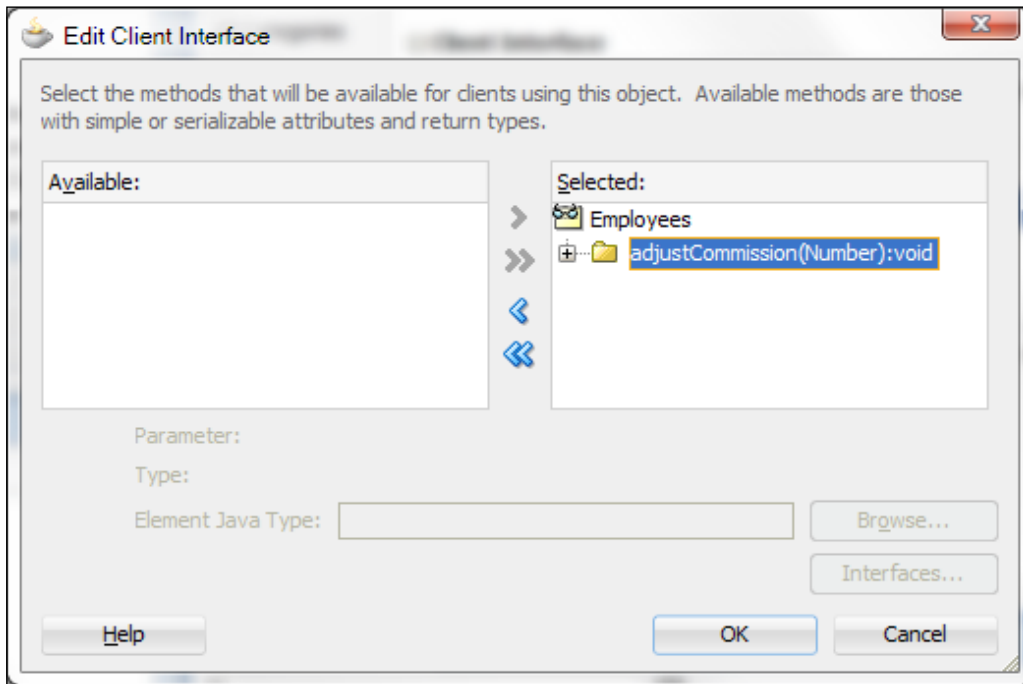
1. Open the `Employees` view object definition and go to the **Java** page.
2. Click on the **Edit java options** button (the pen icon) to open the **Select Java Options** dialog.
3. Click on the **Generate View Object Class** and **Generate View Row Class** checkboxes. Ensure that the **Include accessors** checkbox is also selected.
4. Click **OK** to proceed with the creation of the custom implementation classes.

5. Add the following helper method to `EmployeesImpl.java`. If the import dialog is shown for the `Number` class, make sure that you choose the `oracle.jbo.domain.Number` class.

```
public void adjustCommission(Number commissionPctAdjustment) {
    // check for valid commission adjustment
    if (commissionPctAdjustment != null) {
        // create an employee secondary rowset iterator
        rowsetIterator employees = this.createrowsetIterator(null);
        // reset the iterator
        employees.reset();
        // iterate the employees
        while (employees.hasNext()) {
            // get the employee
            EmployeesRowImpl employee =
                (EmployeesRowImpl)employees.next();
            // check for employee belonging to the sales department
            if (employee.getDepartmentId() != null &&
                SALES_DEPARTMENT_ID ==
                    employee.getDepartmentId().intValue()) {
                // calculate adjusted commission
                Number commissionPct = employee.getCommissionPct();
                Number adjustedCommissionPct = commissionPct != null ?
                    commissionPct.add(commissionPctAdjustment) :
                    commissionPctAdjustment;
                // set the employee's new commission
                employee.setCommissionPct(adjustedCommissionPct);
            }
        }
        // done with the rowset iterator
        employees.closerowsetIterator();
    }
}
```

6. On the **Employees Java** page click on the **Edit view object client interface** button (the pen icon).

7. On the **Edit Client Interface** dialog, shuttle the `adjustCommission()` method to the **Selected** list and click **OK**.



8. Open the `HrComponentsAppModule` application module definition and go to the **Java** page.
9. Click on the **Edit java options** button.
10. On the **Select Java Options** dialog, click on the **Generate Application Module Class** checkbox. Then click **OK** to close the dialog.
11. Open the `HrComponentsAppModuleImpl` class and add the following method:

```
public void adjustCommission(Number commissionPctAdjustment) {  
    // execute the Employees view object query to create  
    // a rowset  
    this.getEmployees().executeQuery();  
    // adjust the employees commission  
    this.getEmployees().adjustCommission(commissionPctAdjustment);  
}
```

12. Return to the application module definition **Java** page, then use the **Edit application module client interface** button to add the `adjustCommission()` method to the application module's client interface.

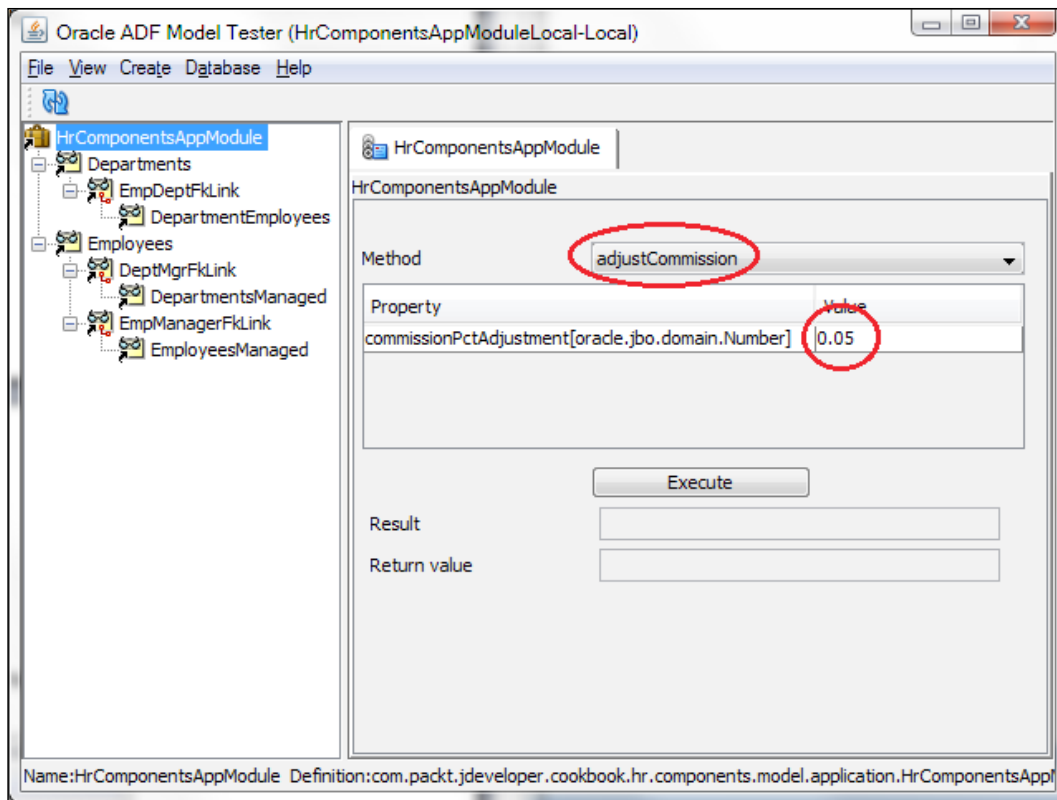
How it works...

We created a view object custom Java implementation class for the `Employees` view object and add a method called `adjustCommission()`. The method is then exposed to the view object's client interface so that it can be accessible and called using the `Employees` interface.

The `adjustCommission()` method adjusts the commission for all employees belonging to the Sales department. The method accepts the commission adjustment percentage as an argument. We call the `createRowsetIterator()` method to create a secondary iterator, which we then use to iterate over the `Employees` rowset. This is the recommended practice to perform programmatic iteration over a rowset. The reason is that the view object instance that is being iterated may be bound to UI components and that iterating it directly will interfere with the UI. In this case, you will see the current row changing by itself.

We then call the `reset()` method to initialize the rowset iterator. This places the iterator in the slot before the first row in the rowset. We iterate the rowset by checking whether a next row exists. This is done by calling `hasNext()` on the iterator. If a next row exists, we retrieve it by calling `next()`, which returns an `oracle.jbo.Row`. We cast the default `Row` object that is returned to an `EmployeesRowImpl`, so we can use the custom setter and getter methods to manipulate the `Employee` row.

For testing purposes, we create a custom application module implementation class and add a method called `adjustCommission()` to it. We expose this method to the application module client interface so that we can call it from the **ADF Model Tester**. Note that methods can also be added to the view object client interface. Then these methods are shown under the view object collection in the **Data Control** panel and can be bound to the JSF page simply by dropping them on the page. Inside the `adjustCommission()`, we execute the `Employees` view object query by calling `executeQuery()` on it. We get the `Employees` view object instance via the `getEmployees()` getter method. Finally, we call the `adjustCommission()` method that we implemented in `EmployeesImpl` to adjust the employees' commission.



There's more...

In order to be able to iterate a view object rowset using a secondary iterator, the view object access mode in the **General | Tuning** section must be set to `Scrollable`. Any other access mode setting will result in a **JBO-25083: Cannot create a secondary iterator on row set {0} because the access mode is forward-only or range-paging** error when attempting to create a secondary iterator. To iterate view objects configured with range paging, use the range paging view object API methods. Specifically, call `getEstimatedRangePageCount()` to determine the number of pages and for each page call `scrollToRangePage()`. Then determine the range page size by calling `getRangeSize()` and iterate through the page calling `getRowAtRangeIndex()`.

Pitfalls when iterating over large rowsets

Before iterating a view object rowset, consider that iterating the rowset may result in fetching a large number of records from the database to the middle layer. In this case, other alternatives should be considered, such as running the iteration asynchronously on a separate **Work Manager**, for instance (see recipe *Using a Work Manager for processing of long running tasks* in *Chapter 12, Optimizing, Fine-tuning and Monitoring*). In certain cases, such as when iterating in order to compute a total amount, consider using any of the following techniques. These methods are far more optimized in determining the total amount for an attribute than iterating the view object using Java code.

- ▶ **Groovy expressions** such as `object.getRowSet().sum('SomeAttribute')`
- ▶ **Analytic functions**, such as `COUNT(args) OVER ([PARTITION BY <...>] ...)`, in the view object's SQL query

For instance, consider the following view object query that calculates the department's total salaries using an analytic function. This would have been much more costly if it had to be done programmatically by iterating the underlying view objects.

```
SELECT DISTINCT DEPARTMENTS.DEPARTMENT_NAME,
SUM (EMPLOYEES.SALARY) OVER (PARTITION BY EMPLOYEES.DEPARTMENT_ID
AS DEPARTMT_SALARIES
FROM EMPLOYEES
INNER JOIN DEPARTMENTS
ON DEPARTMENTS.DEPARTMENT_ID = EMPLOYEES.DEPARTMENT_ID
ORDER BY DEPARTMENTS.DEPARTMENT_NAME
```

See also

- ▶ *Overriding remove() to delete associated children entities, Chapter 2, Dealing with Basics: Entity Objects*

Setting default values for view row attributes

In this recipe, we will see how to set default values for view object attributes. There are a number of places where you can do this, namely:

- ▶ In the overridden `create()` method of the view object row implementation class
- ▶ Declaratively using a Groovy expression
- ▶ In the attribute getter method

For example, for a newly created employee, we will set the employee's hire date to the current date.

Getting ready

This recipe was developed using the `HRComponents` workspace, which was created in the *Overriding `remove()` to delete associated children entities* recipe in *Chapter 2, Dealing with Basics: Entity Objects*. The `HRComponents` workspace requires a database connection to the `HR` schema.

How to do it...

1. Create a view row Java implementation class for the `Employees` view object.
2. Open the `EmployeesRowImpl.java` custom view row Java implementation class and override the `create()` method using the **Override Methods...** button (the green left arrow on the editor toolbar).
3. To set the default employee's hire date to today's date, add the following code to `create()` immediately after the call to `super.create()`:

```
// set the default hire date to today
this.setHireDate((Date)Date.getCurrentDate());
```
4. Open the `Employees` view object definition and go to the **Attributes** page.
5. Select the attribute that you want to initialize, **HireDate** in this case.
6. Select the **Details** tab.
7. In the **Default Value** section, select **Expression** and enter the following Groovy expression: `adf.currentDate`

The screenshot shows the 'Details' tab of the 'Attributes' page in the Oracle ADF IDE. The 'Name' field is 'HireDate' and the 'Display Name' is 'Hire Date'. The 'Type' is 'Date'. Under the 'Default Value' section, the 'Expression' radio button is selected, and the text 'adf.currentDate' is entered in the input field. A red circle highlights the 'Expression' radio button and the text input field.

8. Locate the view object attribute getter in the view object row implementation class. In this example, this is the `getHireDate()` method in `EmployeesRowImpl.java`.
9. Replace the existing code in `getHireDate()` with the following:

```
// get the HireDate attribute value
Date hireDate = (Date)getAttributeInternal(HIREDATE);
// check for null and return today's date if needed
return (hireDate == null) ? (Date)Date.getCurrentDate() :
    hireDate;
```

How it works...

This recipe presents three different techniques to set default values to view object attributes. The first technique (steps 1-3) overrides the view row `create()` method. This method is called by the ADF Business Components framework when a view object row is being created. In the previous code sample, we first call the parent `ViewRowImpl create()` to allow the framework processing. Then we initialize the attribute by calling its setter method—`setHireDate()` in this case—supplying `Date.getCurrentDate()` for the attribute value.

The second technique (steps 4-7) initializes the view object attribute declaratively using a Groovy expression. The Groovy expression used to initialize the `HireDate` attribute is `adf.currentDate`. Note that we change the attribute's **Value Type** field to **Expression**, so that it can be interpreted as an expression instead of a literal value. This expression when evaluated at runtime by the framework retrieves the current date.

Finally, the last technique (steps 8-9) uses the attribute getter—`getHireDate()` for this example—to return a default value. Using this technique, we don't actually set the attribute value; instead we return a default value, which can be subsequently applied to the attribute. Also notice that this is done only if the attribute does not already have a value (the check for null).

There's more...

A common use case related to this topic is setting an attribute's value based on the value of another related attribute. Consider, for instance, the use case where the employee's commission should be set to a certain default value if the employee is part of the Sales department. Also, consider the case where the employee's commission should be cleared if the employee is not part of the sales department. In addition to accomplishing this task with Groovy as stated earlier, it can also be implemented in the employee's `DepartmentId` setter, that is, in the `setDepartmentId()` method as follows:

```
public void setDepartmentId(Number value) {
    // set the department identifier
    setAttributeInternal(DEPARTMENTID, value);
    // set employee's commission based on employee's department
```

```
try {
    // check for Sales department
    if (value != null && SALES_DEPARTMENT_ID == value.intValue()) {
        // if the commission has not been set yet
        if (this.getCommissionPct() == null) {
            // set commission to default
            this.setCommissionPct(new Number(DEFAULT_COMMISSION));
        }
    } else {
        // clear commission for non Sales department
        this.setCommissionPct(null);
    }
} catch (SQLException e) {
    // log the exception
    LOGGER.severe(e);
}
}
```

Specifying default values at the entity object level

Note that default values can be supplied at the entity object level as well. In this case, all view objects based on the particular entity object will inherit the specific behavior. You can provide variations for this behavior by implementing the techniques outlined in this recipe for specific view objects. To ensure consistent behavior throughout the application, it is recommended that you specify attribute defaults at the entity object level.

See also

- ▶ *Overriding `remove()` to delete associated children entities, Chapter 2, Dealing with Basics: Entity Objects*

Controlling the updatability of view object attributes programmatically

In ADF, there are a number of ways to control whether a view object attribute can be updated or not. It can be done declaratively in the **Attributes** tab via the **Updatable** combo, or on the frontend **ViewController** layer by setting the `disabled` or `readOnly` attributes of the **JSF** page component. Programmatically, it can be done either on a backing bean, or if you are utilizing ADF business components, on a custom view object row implementation class. This recipe demonstrates the latter case. For our example, we will disable updating any of the `Department` attributes specifically for departments that have more than a specified number of employees.

Getting ready

This recipe was developed using the `HRComponents` workspace, which was created in the *Overriding `remove()` to delete associated children entities* recipe in *Chapter 2, Dealing with Basics: Entity Objects*. The `HRComponents` workspace requires a database connection to the `HR` schema.

How to do it...

1. Create view row implementation classes for the `Department` and `Employee` view objects. Ensure that in both cases you have selected **Include accessors** on the **Java Options** dialog.
2. Open the `DepartmentsRowImpl` class in the Java editor.
3. Use the **Override Methods...** button to override the `isAttributeUpdateable()` method.
4. Replace the call to `super.isAttributeUpdateable(i)` with the following code:

```
// get the number of employees for the specific department
int departmentEmployeeCount = this.getEmployees() != null
    ? this.getEmployees().getRowCount() : 0;
// set all attributes to non-updatable if the department
// has more than a specified number of employees
return (departmentEmployeeCount > 5)? false :
    super.isAttributeUpdateable(i);
```

How it works...

The `isAttributeUpdateable()` method is called by the framework in order to determine whether a specific attribute is updateable or not. The framework supplies the attribute in question to the `isAttributeUpdateable()` method as an attribute index parameter. Inside the method, we add the necessary code to conditionally enable or disable the specific attribute. We do this by returning a Boolean indicator: a `true` return value indicates that the attribute can be updated.

There's more...

Because the `isAttributeUpdateable()` method could potentially be called several times for each of the view object attributes (when bound to page components for instance), avoid writing code in it that will hinder the performance of the application. For instance, avoid calling database procedures or executing expensive queries in it.

Controlling attribute updatability at the entity object level

Note that we can conditionally control attribute updatability at the entity object level as well, by overriding the `isAttributeUpdateable()` method of `EntityImpl`. In this case, all view objects based on the particular entity object will exhibit the same attribute updatability behavior. You can provide different behavior for specific view objects in this case by overriding `isAttributeUpdateable()` for those objects. To ensure consistent behavior throughout the application, it is recommended that you control attribute updatability defaults at the entity object level.

See also

- ▶ *Overriding `remove()` to delete associated children entities, Chapter 2, Dealing with Basics: Entity Objects*

Setting the Queryable property of a view object attribute programmatically

The `Queryable` property, when set for a view object attribute, indicates that the specific attribute can appear on the view object's `WHERE` clause. This has the effect of making the attribute available in all search forms and allows the user to search for it. In an `af:query` ADF Faces component, for instance, a queryable attribute will appear in the list of fields shown when you click on the **Add Fields** button in the **Advanced** search mode. Declaratively you can control whether an attribute is queryable or not by checking or un-checking the **Queryable** checkbox in the view object **Attributes | Details** tab. But how do you accomplish this task programmatically and for specific conditions?

This recipe will show how to determine the `Queryable` status of an attribute and change it if needed based on a particular condition.

Getting ready

You will need to have access to the shared components workspace that was developed in the *Breaking up the application in multiple workspaces* recipe in *Chapter 1, Pre-requisites to Success: ADF Project Setup and Foundations*. The functionality will be added to the `ExtViewObjectImpl` custom framework class that was developed in the *Setting up BC base classes* recipe in *Chapter 1, Pre-requisites to Success: ADF Project Setup and Foundations*.

How to do it...

1. Open the `ExtViewObjectImpl` view object custom framework class in the Java editor.

2. Add the following method to it:

```
protected void setQueryable(int attribute, boolean condition) {
    // get the attribute definition
    AttributeDef def = getAttributeDef(attribute);
    // set/unset only if needed
    if (def != null && def.isQueryable() != condition) {
        // set/unset queryable
        ViewAttributeDefImpl attributeDef = ViewAttributeDefImpl(def);
        attributeDef.setQueryable(condition);
    }
}
```

How it works...

We have added the `setQueryable()` method to the `ExtViewObjectImpl` view object custom framework class. This makes the method available to all view objects. The method accepts the specific attribute index (`attribute`) and a Boolean indicator whether to set or unset the `Queryable` flag (`condition`) for the specific attribute.

In `setQueryable()`, we first call `getAttributeDef()` to retrieve the `oracle.jbo.AttributeDef` attribute definition. Then we call `isQueryable()` on the attribute definition to retrieve the `Queryable` condition. If the attribute's current `Queryable` condition differs from the one we have passed to `setQueryable()`, we call `setQueryable()` on the attribute definition to set the new value.

Here is an example of calling `setQueryable()` from an application module method based on some attribute values:

```
public void prepare(boolean someCondition) {
    // make the EmployeeId queryable based on some condition
    this.getEmployees().setQueryable(EmployeesRowImpl.EMPLOYEEID,
    someCondition);
}
```

There's more...

Note that you can control the `Queryable` attribute at the entity object level as well. In this case, all view objects based on the specific entity object will inherit this behavior. This behavior can be overridden declaratively or programmatically for the view object, as long as the new value is more restrictive than the inherited value.

See also

- ▶ *Breaking up the application in multiple workspaces, Chapter 1, Pre-requisites to Success: ADF Project Setup and Foundations*
- ▶ *Setting up BC base classes, Chapter 1, Pre-requisites to Success: ADF Project Setup and Foundations*

Using a transient attribute to indicate a new view object row

For entity-based view objects, there is a simple technique you can use to determine whether a particular row has a new status. The status of a row is new when the row is first created. The row remains in the new state until it is successfully committed to the database. It then goes to an unmodified state. Knowledge of the status of the row can be used to set up enable/disable conditions on the frontend user interface.

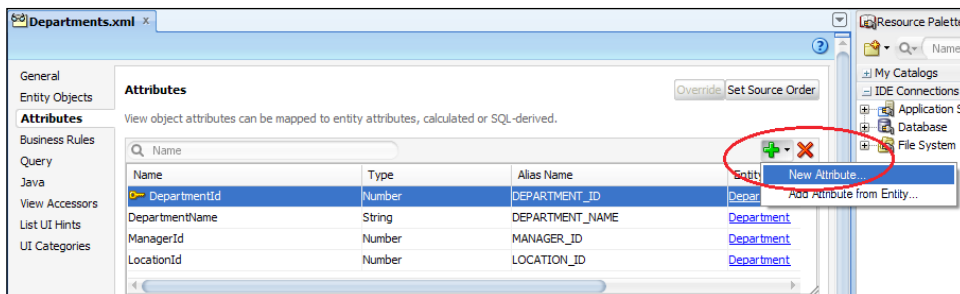
In this recipe, we will see how to utilize a transient view object attribute to indicate the new status of the view object row.

Getting ready

This recipe was developed using the `HRComponents` workspace, which was created in the *Overriding `remove()` to delete associated children entities* recipe in *Chapter 2, Dealing with Basics: Entity Objects*. The `HRComponents` workspace requires a database connection to the HR schema.

How to do it...

1. Open the `Departments` view object definition.
2. Go to the **Attributes** tab and click on the **Create new attribute** button (the green plus sign icon).
3. Select **New Attribute...** from the context menu.



4. On the **New View Object Attribute** dialog, enter `IsNewRow` and click **OK**.
5. By default the new attribute is of type `String`, so change it to a `Boolean` using the **Type** choice list in the **Details** tab.
6. If you don't already have a custom view row implementation class created, use the **Java** tab to create one. In any case, make sure that you have selected the **Include accessors** checkbox.
7. Open the `DepartmentsRowImpl.java` view row implementation class and locate the `getIsNewRow()` method.
8. Replace the code inside the `getIsNewRow()` method with the following:

```
// return true if the row status is New
return Row.STATUS_NEW == this.getDepartment().getEntityState();
```

How it works...

First we create a new transient attribute called `IsNewRow`. This attribute will be used to indicate whether the status of the view object row is new or not. A transient attribute is one that does not correspond to a database table column; it can be used as placeholder for intermediate data. Then we generate a custom view row implementation class. On the transient attribute getter, `getIsNewRow()` in this case, we get access to the entity object. For this recipe, the `Department` entity is returned by calling the `getDepartment()` getter. We get the entity object state by calling `getEntityState()` on the `Department` entity object and compare it to the constant `Row.STATUS_NEW`.

Once the `IsNewRow` attribute is bound to a JSF page, it can be used in **Expression Language (EL)** expressions. For instance, the following EL expression indicates a certain disabled condition based on the row status not being New:

```
disabled="#{bindings.IsNewRow.inputValue ne true}"
```

There's more...

The following table summarizes all the available entity object states:

Entity Object State	Description	Transition to this State
New	Indicates a new entity object. An entity object in this state is in the transaction pending changes list (see <i>Initialized</i> state).	When a new entity object is first created. When <code>setAttribute()</code> is called on an <i>Initialized</i> entity object.
Initialized	Indicates that a new entity object is initialized and thus it is removed from the transaction's pending changes list.	When <code>setNewRowState()</code> is explicitly called on a New entity object.

Entity Object State	Description	Transition to this State
Unmodified	Indicates an unmodified entity object.	When the entity object is retrieved from the database. After successfully committing a New or Modified entity object.
Modified	Indicates the state of a modified entity object.	When <code>setAttribute()</code> is called on an Unmodified entity object.
Deleted	Indicates a deleted entity object.	When <code>remove()</code> is called on an Unmodified or Modified entity object.
Dead	Indicates a dead entity object.	When <code>remove()</code> is called on a New or Initialized entity object. After successfully committing a Deleted entity object.

See also

- ▶ *Overriding `remove()` to delete associated children entities, Chapter 2, Dealing with Basics: Entity Objects*

Conditionally inserting new rows at the end of the rowset

When you insert a new row into a rowset, by default the new row is inserted at the current slot within that rowset. There are times, however, that you want to override this default behavior for the application that you are developing.

In this recipe, we will see how to conditionally insert new rows at the end of the rowset by implementing generic programming functionality at the base view object framework implementation class.

Getting ready

You will need to have access to the shared components workspace that was developed in the *Breaking up the application in multiple workspaces* recipe in *Chapter 1, Pre-requisites to Success: ADF Project Setup and Foundations*. The functionality will be added to the `ExtViewObjectImpl` custom framework class that was developed in the *Setting up BC base classes* recipe in *Chapter 1, Pre-requisites to Success: ADF Project Setup and Foundations*.

How to do it...

1. Open the `ExtViewObjectImpl.java` custom view object framework class in the Java editor.
2. Override the `insertRow()` method.
3. Replace the call to `super.insertRow()` in the generated `insertRow()` method with the following code:

```
// check for overridden behavior based on custom property
if ("true".equalsIgnoreCase((String)this.getProperty(
    NEW_ROW_AT_END))) {
    // get the last row in the rowset
    Row lastRow = this.last();
    if (lastRow != null) {
        // get index of last row
        int lastRowIdx = this.getRangeIndexof(lastRow);
        // insert row after the last row
        this.insertRowAtRangeIndex(lastRowIdx + 1, row);
        // set inserted row as the current row
        this.setCurrentRow(row);
    } else {
        super.insertRow(row);
    }
} else {
    // default behavior: insert at current rowset slot
    super.insertRow(row);
}
```

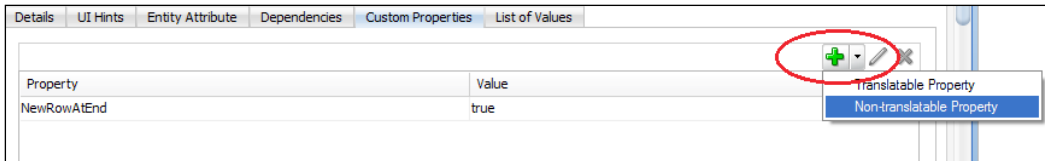
How it works...

We have overridden the ADF Business Components framework `insertRow()` method in order to implement custom row insertion behavior. Moreover, we conditionally override the default framework behavior based on the existence of the **custom property** `NewRowAtEnd` identified by the constant `NEW_ROW_AT_END`. So, if this custom property is defined for specific view objects, we determine the index of the last row in the rowset by calling `getRangeIndexof()` and then call `insertRowAtRangeIndex()` to insert the new row at the specific last row index. Finally, we set the rowset currency to the row just inserted.

If the `NewRowAtEnd` custom property is not defined in the view object, then the row is inserted by default at the current slot in the rowset.

There's more...

To add a custom property to a view object, use the drop-down menu next to the **Add Custom Property** button (the green plus sign icon) and select **Non-translatable Property**. The **Add Custom Property** button is located in the **Attributes | Custom Properties** tab.



In addition, note that if you have configured range paging access mode for the view object, calling `last()` will produce a **JBO-25084: Cannot call last() on row set {0} because the access mode uses range-paging** error. In this case, call `getEstimatedRangePageCount()` to determine the number of pages and `setRangeStart()` to set the range to the last page instead.

Inserting new rows at the beginning of the rowset

The use case presented in this recipe can be easily adapted to insert a row at the beginning of the rowset. In this case, you will need to call `this.first()` to get the first row. The functionality of getting the row index and inserting the row at the specified index should work as is.

See also

- ▶ *Breaking up the application in multiple workspaces, Chapter 1, Pre-requisites to Success: ADF Project Setup and Foundations*
- ▶ *Setting up BC base classes, Chapter 1, Pre-requisites to Success: ADF Project Setup and Foundations*

Using `findAndSetCurrentRowByKey()` to set the view object currency

You can set the currency on a view object by calling its `findAndSetCurrentRowByKey()` method. The method accepts two arguments: a `Key` object that is used to locate the row in the view object, and an integer indicating the range position of the row (for view objects configured with range paging access mode).

This recipe demonstrates how to set the view object row currency by implementing a helper method called `refreshView()`. In it we first save the view object currency, re-query the view object and finally restore its currency to the original row before the re-query. This has the effect of refreshing the view object while keeping the current row.

Getting ready

You will need to have access to the shared components workspace that was developed in the *Breaking up the application in multiple workspaces* recipe in *Chapter 1, Pre-requisites to Success: ADF Project Setup and Foundations*. The functionality will be added to the `ExtViewObjectImpl` custom framework class that was developed in the *Setting up BC base classes* recipe in *Chapter 1, Pre-requisites to Success: ADF Project Setup and Foundations*.

How to do it...

1. Open the `ExtViewObjectImpl.java` view object framework extension class in the Java editor.
2. Override the `create()` method.
3. Add the following code after the call to `super.create()`:


```
// allow read-only view objects to use findByKey() methods
this.setManageRowsByKey(true);
```
4. While at the `ExtViewObjectImpl.java` add the following `refreshView()` method:

```
public void refreshView() {
    Key curRowKey = null;
    int rangePosOfCurRow = -1;
    int rangeStart = -1;
    // get and save the current row
    Row currentRow = getCurrentRow();
    // do this only if we have a current row
    if (currentRow != null) {
        // get the row information
        curRowKey = currentRow.getKey();
        rangePosOfCurRow = getRangeIndexOf(currentRow);
        rangeStart = getRangeStart();
    }
    // execute the view object query
    executeQuery();
    // if we have a current row, restore it
    if (currentRow != null) {
        setRangeStart(rangeStart);
        findAndSetCurrentRowByKey(curRowKey, rangePosOfCurRow);
    }
}
```

How it works...

First we override the `create()` method in the view object framework extension class. We do this so we can call the `setManageRowsByKey()` method. This method will allow us to use framework find methods utilizing key objects on read-only view objects as well. By default this is not the case for read-only view objects.

Then we implement the `refreshView()` method. We made `refreshView()` public so that it could be called explicitly for all view objects. Once bound to a page definition as an operation binding, `refreshView()` can be called from the UI. You can, for instance, include a refresh button on your UI page, which when pressed refreshes the data presented in a table. In it, we first determine the current row in the rowset by calling `getCurrentRow()`. This method returns an `oracle.jbo.Row` object indicating the current row, or `null` if there is no current row in the rowset. If there is a current row, we get all the necessary information in order to be able to restore it after we re-query the view object. This information includes the current row's key (`getKey()`), the index of the current row in range (`getRangeIndexOf()`), and the row's range (`getRangeStart()`).

Once the current row information is saved, we re-execute the view object's query by calling `executeQuery()`.

Finally, we restore the current row by setting the range and the row within the range by calling `setRangeStart()` and `findAndSetCurrentRowByKey()` respectively.

There's more...

The methods `getRangeIndexOf()`, `getRangeStart()`, and `setRangeStart()` used in this recipe indicate that range paging optimization is utilized.

Range paging optimization

Range paging is an optimization technique that can be utilized by view objects returning large result sets. The effect of using it is to limit the result set to a specific number of rows, as determined by the **range size** option setting. So instead of retrieving hundreds or even thousands of rows over the network and caching them in the middle layer memory, only the ranges of rows utilized are retrieved.



The methods used in this recipe to retrieve the row information will work regardless of whether we use range paging or not.

For more information on this recipe consult Steve Muench's original post *Refreshing a View Object's Query, Keeping Current Row and Page* in the URL address: <http://radio-weblogs.com/0118231/2004/11/22.html>.

See also

- ▶ *Breaking up the application in multiple workspaces, Chapter 1, Pre-requisites to Success: ADF Project Setup and Foundations*
- ▶ *Setting up BC base classes, Chapter 1, Pre-requisites to Success: ADF Project Setup and Foundations*
- ▶ *Restoring the current row after a transaction rollback, Chapter 2, Dealing with Basics: Entity Objects*

Restoring the current row after a transaction rollback

On a transaction rollback, the default behavior of the ADF framework is to set the current row to the first row in the rowset. This is certainly not the behavior you expect to see when you rollback while editing a record.

This recipe shows how to accomplish the task of restoring the current row after a transaction rollback.

Getting ready

You will need to have access to the shared components workspace that was developed in the *Breaking up the application in multiple workspaces* recipe in *Chapter 1, Pre-requisites to Success: ADF Project Setup and Foundations*. The functionality will be added to the `ExtApplicationModuleImpl` and `ExtViewObjectImpl` custom framework classes that were developed in the *Setting up BC base classes* recipe in *Chapter 1, Pre-requisites to Success: ADF Project Setup and Foundations*.

How to do it...

1. Open the `ExtApplicationModuleImpl.java` application module framework extension class into the Java editor.
2. Click on the **Override Methods...** button (the green left arrow button) and choose to override the `prepareSession(Session)` method.
3. Add the following code after the call to `super.prepareSession()`:

```
// do not clear the cache after a rollback
getDBTransaction().setClearCacheOnRollback(false);
```
4. Open the `ExtViewObjectImpl.java` view object framework extension class into the Java editor.

5. Override the `create()` method.

6. Add the following code after the call to `super.create()`:

```
// allow read-only view objects to use findByKey() methods
this.setManageRowsByKey(true);
```

7. Override the `beforeRollback()` method.

8. Add the following code before the call to `super.beforeRollback()`:

```
// check for query execution
if (isExecuted()) {
    // get the current row
    ViewRowImpl currentRow = (ViewRowImpl)getCurrentRow();
    if (currentRow != null) {
        // save the current row's key
        currentRowKeyBeforeRollback = currentRow.getKey();
        // save range start
        rangeStartBeforeRollback = getRangeStart();
        // get index of current row in range
        rangePosOfCurrentRowBeforeRollback =
            getRangeIndexOf(currentRow);
    }
}
```

9. Override the `afterRollback()` method and add the following code after the call to `super.afterRollback()`:

```
// check for current row key to restore
if (currentRowKeyBeforeRollback != null) {
    // execute view object's query
    executeQuery();
    // set start range
    setRangeStart(rangeStartBeforeRollback);
    // set current row in range
    findAndSetCurrentRowByKey(currentRowKeyBeforeRollback,
        rangePosOfCurrentRowBeforeRollback);
}
// reset
currentRowKeyBeforeRollback = null;
```

How it works...

We override the application module `prepareSession()` method so we can call `setClearCacheOnRollback()` on the `Transaction` object. `prepareSession()` is called by the ADF Business Components framework the first time that an application module is accessed. The call to `setClearCacheOnRollback()` in `prepareSession()` tells the framework whether the entity cache will be cleared when a rollback operation occurs. Since the framework by default clears the cache, we call `setClearCacheOnRollback()` with a `false` argument to prevent this from happening. We need to avoid clearing the cache because we will be getting information about the current row during the rollback operation.

Whether to clear the entity cache on transaction rollback or not is a decision that needs to be taken at the early design stages, as this would affect the overall behavior of your ADF application. For the specific technique in this recipe to work, that is, to be able to maintain the current row after a transaction rollback, the entity cache must not be cleared after a transaction rollback operation. If this happens, fresh entity rows for the specific entity type will be retrieved from the database, which prevents this recipe from working properly.

Next, we override the `create()` method in the view object framework extension class. We do this so we can call the `setManageRowsByKey()` method. This method will allow us to use framework find methods utilizing key objects on read-only view objects, which is not the default behavior.

Then we override the `beforeRollback()` and `afterRollback()` methods in the view object framework extension class. As their names indicate, they are called by the framework before and after the actual rollback operation. Let's take a look at the code in `beforeRollback()` first. In it, we first get the current row by calling `getCurrentRow()`. Then, for the current row we determine the row's key, range, and position of the current row within the range. This will work whether range paging is used for the view object or not. Range paging should be enabled for optimization purposes for view objects that may return large rowsets. We save these values into corresponding member variables. We will be using them in `afterRollback()` to restore the current row after the rollback operation.



Notice that we do all that after checking whether the view object query has been executed (`isExecuted()`). We do this because `beforeRollback()` may be called multiple times by the framework, and we need to ensure that we retrieve the current row information only if the view object's rowset has been updated, which is the case after the query has been executed.

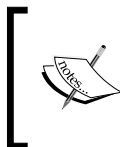
In `afterRollback()` we use the information obtained about the current row in `beforeRollback()` to restore the rowset currency to it. We do this by first executing the view object query, the call to `executeQuery()`, and then calling `setRangeStart()` and `findAndSetCurrentRowByKey()` to restore the range page and the row within the range to the values obtained for the current row earlier. We do all that only if we have a current row key to restore to – the check for `currentRowKeyBeforeRollback` not being null.

There's more...

Note that for this technique to work properly on the frontend `ViewController`, you will need to call `setExecuteOnRollback()` on the `oracle.adf.model.binding.DCBindingContainer` object before executing the Rollback operation binding. By calling `setExecuteOnRollback()` on the binding container, we prevent the view objects associated with the page's bindings being executed after the rollback operation. This means that in order to call `setExecuteOnRollback()` you can't just execute the rollback action binding directly. Rather, you will have to associate the Rollback button with a backing bean action method, similar to the one shown in the following instance:

```
public void rollback() {
    // get the binding container
    DCBindingContainer bindings = ADFUtils.getDCBindingContainer();
    // prevent view objects from executing after rollback
    bindings.setExecuteOnRollback(false);
    // execute rollback operation
    ADFUtils.findOperation("Rollback").execute();
}
```

Such a method could be added to the `CommonActions` generic backing bean actions framework class introduced in the recipe *Using a generic backing bean actions framework* in *Chapter 1, Pre-requisites to Success: ADF Project Setup and Foundations*. By doing so, we will make it available to all managed beans derived from the `CommonActions` base class.



For testing purposes, a `ViewController` project page called `recipe3_8.jspx` has been provided that demonstrates this technique. To run it, just right-click on the **recipe3_8.jspx** page and select **Run** from the context menu.

For more information on this technique, consult Steve Muench's original post *Example of Restoring Current Row On Rollback* (example number 68) on the URL address: <http://radio-weblogs.com/0118231/2006/06/15.html>.

See also

- ▶ *Breaking up the application in multiple workspaces, Chapter 1, Pre-requisites to Success: ADF Project Setup and Foundations*
- ▶ *Setting up BC base classes, Chapter 1, Pre-requisites to Success: ADF Project Setup and Foundations*
- ▶ *Using `findAndSetCurrentRowByKey()` to set the view object currency, Chapter 2, Dealing with Basics: Entity Objects.*
- ▶ *Overriding `prepareSession()` to do session-specific initializations, Chapter 5, Putting them all together: Application Modules*

Dynamically changing the WHERE clause of the view object query

During the execution of the view object's query, the ADF Business Components framework calls a series of methods to accomplish its task. You can intervene during this process by overriding any of the methods called by the framework, in order to change the query about to be executed by the view object. You can also explicitly call methods in the public view object interface to accomplish this task prior to the view object's query execution. Depending on what exactly you need to change in the view object's query, the framework allows you to do the following:

- ▶ Change the query's `SELECT` clause by overriding `buildSelectClause()` or calling `setSelectClause()`
- ▶ Change the query's `FROM` clause by overriding `buildFromClause()` or calling `setFromClause()`
- ▶ Change the query's `WHERE` clause via `buildWhereClause()`, `setWhereClause()`, `addWhereClause()`, `setWhereClauseParams()`, and other methods
- ▶ Change the query's `ORDER BY` clause via the `buildOrderByClause()`, `setOrderByClause()`, and `addOrderByClause()` methods

Even the complete query can be changed by overriding the `buildQuery()` method or directly calling `setQuery()`. Moreover, adding named view criteria will alter the view object query.

This recipe shows how to override `buildWhereClause()` to alter the view object's `WHERE` clause. The use case implemented in this recipe is to limit the result set of the `Employee` view object by a pre-defined number of rows indicated by a custom property.

Getting ready

You will need to have access to the shared components workspace that was developed in the *Breaking up the application in multiple workspaces* recipe in *Chapter 1, Pre-requisites to Success: ADF Project Setup and Foundations*. The functionality will be added to the `ExtViewObjectImpl` custom framework class that was developed in the *Setting up BC base classes* recipe in *Chapter 1, Pre-requisites to Success: ADF Project Setup and Foundations*.

How to do it...

1. Open the `ExtViewObjectImpl.java` view object framework extension class in the Java editor.
2. Override the `buildWhereClause()` method.
3. Replace the code inside the `buildWhereClause()` with the following:

```
// framework processing
boolean appended = super.buildWhereClause(sqlBuffer,noUserParams);
// check for a row count limit
String rowCountLimit = (String)this.getProperty(ROW_COUNT_LIMIT);
// if a row count limit exists, limit the query
if (rowCountLimit != null) {
    // check to see if a WHERE clause was appended;
    // if not, we will append it
    if (!appended) {
        // append WHERE clause
        sqlBuffer.append(" WHERE ");
        // indicate that a where clause was added
        appended = true;
    }
    // a WHERE clause was appended by the framework;
    // just amend it
    else {
        sqlBuffer.append(" AND ");
    }
    // add ROWNUM limit based on the pre-defined
    // custom property
    sqlBuffer.append("(ROWNUM <= " + rowCountLimit + ")");
}
// a true/false indicator whether a WHERE clause was appended
// is returned to the framework
return appended;
```

How it works...

We override `buildWhereClause()` in order to alter the the `WHERE` clause of the view object's query. Specifically, we limit the result set produced by the view object's query. We do this only if the custom property called `RowCountLimit` (indicated by the constant `ROW_COUNT_LIMIT`) is defined by a view object. The value of the `RowCountLimit` indicates the number of rows that the view object's result set should be limited to.

First we call `super.buildWhereClause()` to allow the framework processing. This call will return a Boolean indicator of whether a `WHERE` clause was appended to the query, indicated by the Boolean `appended` local variable. Then we check for the existence of the `RowCountLimit` custom property. If it is defined by the specific view object, we alter or we add to the `WHERE` clause depending on whether one was added or not by the framework. We make sure that we set the `appended` flag to `true` if we actually have appended the `WHERE` clause. Finally, the `appended` flag is returned back to the framework.

There's more...

The use case implemented in this recipe shows one of the possible ways of limiting the view object result set. You can explore additional techniques in *Chapter 12, Optimizing, Fine-tuning and Monitoring*.

See also

- ▶ *Breaking up the application in multiple workspaces, Chapter 1, Pre-requisites to Success: ADF Project Setup and Foundations*
- ▶ *Setting up BC base classes, Chapter 1, Pre-requisites to Success: ADF Project Setup and Foundations*

Removing a row from a rowset without deleting it from the database

There are times when you want to remove a row from the view object's query collection (the query result set) without actually removing it from the database. The query collection, `oracle.jbo.server.QueryCollection`, gets populated each time the view object is executed—when the view object's associated query is run—and represents the query result. While the `Row.remove()` method will remove a row from the query collection, it will also remove the underlying entity object for an entity-based view object, and post a deletion to the database. If your programming task requires the row to be removed from the query collection itself, that is, removing a table row in the UI without actually posting a delete to the database, use the `Row` method `removeFromCollection()` instead.



Note, however, that each time the view object query is re-executed the row will show up, since it is not actually deleted from the database.

This recipe will demonstrate how to use `removeFromCollection()` by implementing a helper method in the application module to remove rows from the `Employees` collection.

Getting ready

This recipe was developed using the `HRComponents` workspace, which was created in the *Overriding `remove()` to delete associated children entities* recipe in *Chapter 2, Dealing with Basics: Entity Objects*. The `HRComponents` workspace requires a database connection to the `HR` schema.

How to do it...

1. Open the `HrComponentsAppModuleImpl.java` application module custom implementation class in the Java editor.
2. Add the following `removeEmployeeFromCollection()` method to it:

```
public void removeEmployeeFromCollection() {
    // get the current employee
    EmployeesRowImpl employee =
        (EmployeesRowImpl)(this.getEmployees().getCurrentRow());
    // remove employee from collection
    if (employee != null) {
        employee.removeFromCollection();
    }
}
```
3. Expose the `removeEmployeeFromCollection()` method to the application module's client interface using the **Edit application module client interface** button (the pen icon) in the **Client Interface** section of the application module's **Java** page.

How it works...

We implemented `removeEmployeeFromCollection()` and exposed it to the application module's client interface. By doing so, we will be able to call this method using the **Oracle ADF Model Tester** for testing purposes.

First, we get the current employee by calling `getCurrentRow()` on the `Employees` view object instance. We retrieve the `Employees` view object instance by calling the `getEmployees()` getter method. Then we call `removeFromCollection()` on the current employee view row to remove it from the `Employees` view object rowset. This has the effect of removing the employee from the rowset without removing the employee from the database. A subsequent re-query of the `Employees` view object will retrieve those `Employee` rows that were removed earlier.

There's more...

Note that there is a quick and easy way to remove all rows from the view object's rowset by calling the view object `executeEmptyrowset()` method. This method re-executes the view object's query ensuring that the query will return no rows; however, it achieves this in an efficient programmatic way without actually sending the query to the database for execution. Calling `executeEmptyrowset()` marks the query's `isExecuted` flag to `true`, which means that it will not be re-executed upon referencing a view object attribute.

See also

- ▶ *Overriding `remove()` to delete associated children entities, Chapter 2, Dealing with Basics: Entity Objects*

4

Important Contributors: List of Values, Bind Variables, View Criteria

In this chapter, we will cover:

- ▶ Setting up multiple LOVs using a switcher attribute
- ▶ Setting up cascading LOVs
- ▶ Creating static LOVs
- ▶ Overriding `bindParametersForCollection()` to set a view object bind variable
- ▶ Creating view criteria programmatically
- ▶ Clearing the values of bind variables associated with the view criteria
- ▶ Searching case-insensitively using view criteria

Introduction

List of values (LOV), bind variables and **view criteria** are essential elements related to view objects. They allow further refinements to the view object's query (bind variables and view criteria) and make the development of the frontend user interface easier when dealing with list controls (LOVs) and query-by-example (view criteria) components.

Many of the user interface aspects that deal with list controls and query-by-example components can be pre-defined in a set of default values via the **UI Hints** sections and pages in JDeveloper, thus providing a standard UI behavior. By using LOVs, for instance, we can pre-define a number of attributes for UI list components, such as the default UI list component, the attributes to be displayed, whether "No Selection" items will be included in the list, and others. These defaults can be overridden as needed for specific LOVs.

Bind variables and view criteria, usable in conjunction or separately, allow you to dynamically alter the view object query based on certain conditions. Furthermore, using bind variables as placeholders in the query allows the database to effectively reuse the same parsed query for multiple executions, without the need to re-parse it.

In this chapter, we will examine how these components are supported, both programmatically and declaratively, by the ADF-BC framework and by JDeveloper.

Setting up multiple LOVs using a switcher attribute

Enabling LOVs for view object attributes greatly simplifies the effort involved in utilizing list controls in the frontend user interface. LOV-enabling a view object attribute is a straightforward task, done declaratively in JDeveloper. Moreover, the ADF-BC framework allows you to define multiple LOVs for the same attribute. In this case, in order to differentiate among the LOVs, a separate attribute called a LOV switcher is used. The differentiation is usually done based on some data value. The advantage of using this technique is that you can define a single LOV component in your UI page and then vary its contents based on a certain condition, such as the value of the switcher attribute.

This recipe shows how to enable multiple LOVs for a view object attribute and how to use an LOV switcher to switch among the LOVs. For example, depending on the employee's job we will associate a different LOV to an `Employees view object` transient attribute.

Getting ready

This recipe was developed using the `HRComponents` workspace, which was created in the *Overriding `remove()` to delete associated children entities* recipe in *Chapter 2, Dealing with Basics: Entity Objects*. The `HRComponents` workspace requires a database connection to the HR schema.

How to do it...

1. Create a new read-only view object called `DepartmentsLov` by right-clicking on a package of the `HRComponents` business components project in the **Application Navigator** and selecting **New View Object...**
2. Base the view object on the following SQL query:

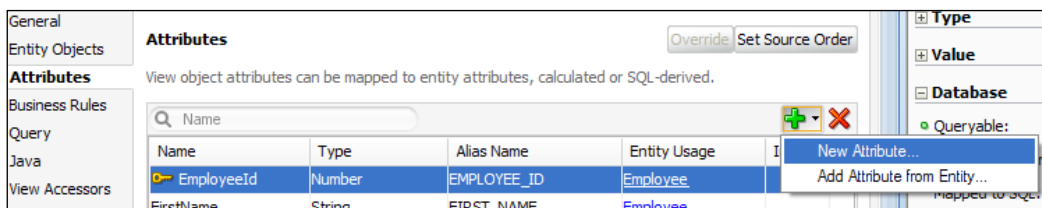

```
SELECT DEPARTMENT_ID, DEPARTMENT_NAME FROM DEPARTMENTS
```

In addition, set the `DepartmentId` attribute as the a **Key attribute**. Do not add the `DepartmentsLov` view object to the application module.
3. Repeat the previous steps to create another read-only view object called `JobsLov` based on this SQL query:


```
SELECT JOB_ID, JOB_TITLE FROM JOBS
```
4. In this case, set the `JobId` attribute as the key attribute.
5. Create yet another read-only view object called `CountriesLov` based on the following SQL query:


```
SELECT COUNTRY_ID, COUNTRY_NAME FROM COUNTRIES
```

Define `CountryId` as the key attribute.
6. Now, open the `Employees` view object definition and go to the **Attributes** tab. Create a new attribute called `LovAttrib` by selecting **New Attribute...** from the context menu.



7. In the **Details** tab, change the attribute **Updatable** value to **Always**.
8. Switch to the **List of Values** tab and click on the **Add list of values** button (the green plus sign icon).
9. On the **Create List of Values** dialog, enter `LOV_Departments` for the **List of Values Name**.
10. Click on the **Create new view accessor** button (the green plus sign icon next to the **List Data Source**) to create a new **List Data Source**. This will bring up the **View Accessors** dialog.
11. On the **View Accessors** dialog, locate the `DepartmentsLov`, shuttle it to the **View Accessors** list on the right and click **OK**.

12. Select the `DepartmentName` for the **List Attribute**. In the **List Return Values** section, the `DepartmentName` view accessor attribute should be associated with the `LovAttrib`. Click **OK** when done.
13. Repeat the previous steps to add another LOV, called `LOV_Jobs`. Add the `JobsLov` as a view accessor, as you did in the previous steps, and select it as the **List Data Source**. Use the `JobTitle` attribute as the **List Attribute**.
14. Add one more LOV called `LOV_Countries` by repeating the previous steps. Add `CountriesLov` as a view accessor and select it as the **List Data Source**. For the **List Attribute**, use the `CountryName` attribute.
15. While at the **List of Values** tab, click on the **Create new attribute** button (the green plus sign icon) next to the **List of Values Switcher** field to create a switcher attribute. Call the attribute `LovSwitcher`. Now, the **List of Values** tab should look similar to the following:

Default	Name	List Data Source	List Attribute
<input checked="" type="radio"/>	LOV_Departments	DepartmentsLov	DepartmentId
<input type="radio"/>	LOV_Jobs	JobsLov	JobId
<input type="radio"/>	LOV_Countries	CountriesLov	CountryId

16. Select the `LovSwitcher` attribute and go to the **Details** tab. Click on the **Expression** radio button in the **Default Value** section, and then on the **Edit value** button (the pen icon). Enter the following expression:

```

if(JobId == 'SA_REP'){
    return 'LOV_Countries'
} else if(JobId == 'ST_CLERK'){
    return 'LOV_Jobs'
} else if(JobId == 'ST_MAN'){
    return 'LOV_Departments'
} else {
    return null;
}
    
```

17. Optionally click on the **UI Hints** tab and change the **Display Hint** from **Display** to **Hide**.

How it works...

In steps 1 through 4 we created three read-only view objects, one for each LOV. We did not add any of these read-only view objects to the application module's data model as they are used internally by the business service. We then created a new transient attribute, called `LovAttrib`, for the `Employees` view object (step 5). This is the attribute that we will use to add the three LOVs. We added the LOVs by switching to the **List of Values** tab. In steps 6 through 13, we added the appropriate view objects as accessors to the `Employees` view object and associated the accessors with the LOV **List Data Source**. This indicates the view accessor that provides the list data at runtime. In each case, we also specified a view accessor attribute as the list attribute. This is the view accessor attribute that supplies the data value to the `LovAttrib` attribute. You can specify additional view accessor attributes to supply data for other `Employees` view object attributes in the **List Return Values** section of the **Create/Edit List of Values** dialog. In step 14, we created a new transient attribute, called `LovSwitcher`, to act as the LOV switcher. In step 15, we supplied the default value to the LOV switcher `LovSwitcher` attribute in the form of a Groovy expression. In the Groovy expression, we examine the value of the `JobId` attribute and based on its value we assign (by returning the LOV name) the appropriate LOV to the `LovSwitcher` attribute. Since the `LovSwitcher` attribute is used as an LOV switcher, the result is that the appropriate LOV is associated with the `LovAttrib` attribute. Finally, note that in step 16 you can optionally set the **Display Hint** to **Hide** for the `LovAttrib` attribute. This will ensure that the specific attribute is not visible in the presentation layer UI.

There's more...

For entity-based view objects, you can LOV-enable an attribute using a list data source that is based on an entity object view accessor. This way a single entity-based view accessor is used for all view objects based on the entity object, and is applied to each instance of the LOV. Note, however, that while this will work fine on create and edit forms, it will not work for search forms. In this case, the LOV must be based on a view accessor defined at the view object level. For a use case where two LOVs are defined on an attribute—one based on an entity object accessor and another on a view object accessor—you can use a switcher attribute that differentiates among the two LOVs based on the following expression:

```
adf.isCriteriaRow ? "LOV_ViewObject_accessor" :
"LOV_EntityObject_accessor"
```

For more information on this consult the section *How to Specify Multiple LOVs for an LOV-Enabled View Object Attribute* in the *Fusion Developer's Guide for Oracle Application Development Framework* which can be found at http://docs.oracle.com/cd/E24382_01/web.1112/e16182/toc.htm.

See also

- ▶ *Overriding remove() to delete associated children entities, Chapter 2, Dealing with Basics: Entity Objects*

Setting up cascading LOVs

Cascading LOVs refer to two or more LOVs where the possible values of one LOV depend on specific attribute values defined in another LOV. These controlling attributes are used in order to filter the result set produced by the controlled LOVs. The filtering is usually accomplished by adding named view criteria, based on bind variables, to the controlled LOV list data source (the view accessor). This allows you to filter the view object result set by adding query conditions that augment the view object query `WHERE` clause. Furthermore, the filtering can be done by directly modifying the controlled LOV view accessor query, adding the controlling attributes as bind variable placeholders in its query. This technique comes handy when you want to set up interrelated LOV components in your UI pages, where the contents of one LOV are filtered based on the value selected in the other LOV.

For this recipe, we will create two LOVs, one for the `DEPARTMENTS` table and another for the `EMPLOYEES` table, so that when a department is selected only the employees of that particular department are shown.

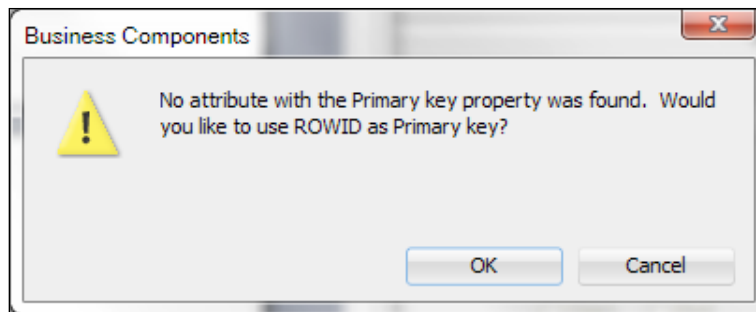
Getting ready

This recipe was developed using the `HRComponents` workspace, which was created in the *Overriding remove() to delete associated children entities* recipe in *Chapter 2, Dealing with Basics: Entity Objects*. The `HRComponents` workspace requires a database connection to the `HR` schema. You will also need an additional table added to the `HR` schema called `CASCADING_LOVS`. You can create it by running the following SQL command:

```
CREATE TABLE CASCADING_LOVS (EMPLOYEE_ID NUMBER(6), DEPARTMENT_ID
NUMBER(4));
```

How to do it...

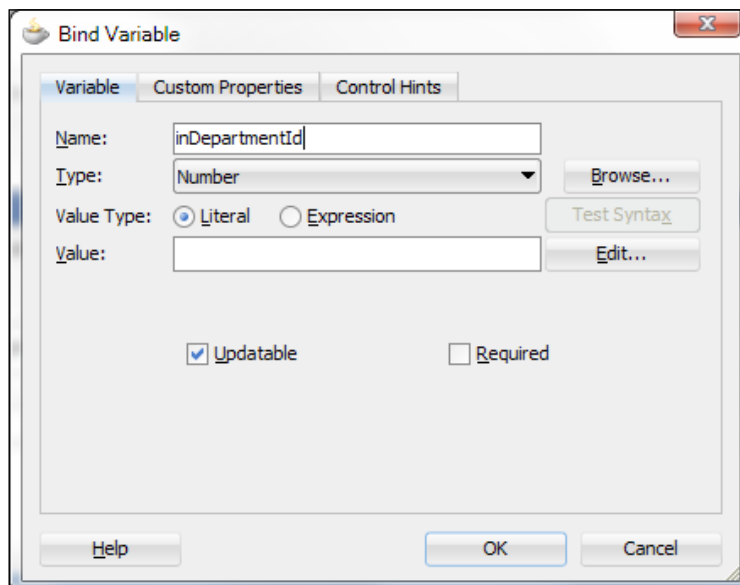
1. Create a new entity object based on the `CASCADING_LOVS` table.
2. Since the `CASCADING_LOVS` table does not define a primary key, the **Create Entity Object** wizard will ask you if you want to create an attribute with a primary key property based on the `ROWID`. Select **OK**.



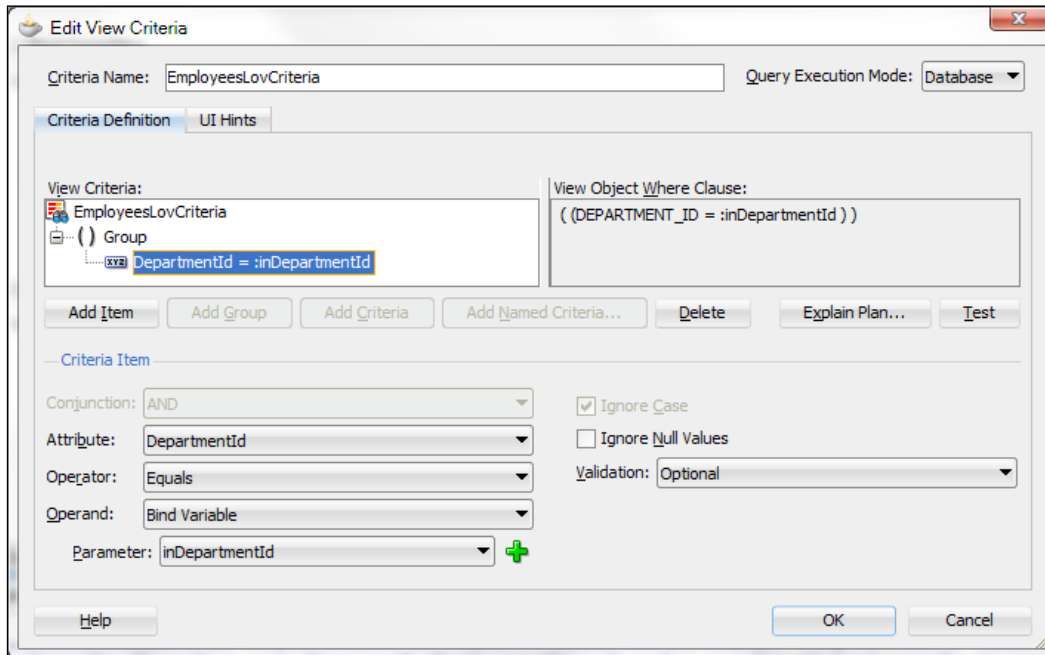
3. Create a view object based on the `CascadingLovs` entity object.
4. Create a read-only view object called `DepartmentsLov` based on the following query:


```
SELECT DEPARTMENT_ID, DEPARTMENT_NAME FROM DEPARTMENTS
```
5. Create another read-only view object called `EmployeesLov` based on the following query:


```
SELECT DEPARTMENT_ID, EMPLOYEE_ID, FIRST_NAME, LAST_NAME
      FROM EMPLOYEES
```
6. In the **Query** section, use the **Create new bind variable** button (the green plus sign icon) to add a bind variable to `EmployeesLov`. Call the bind variable `inDepartmentId` of **Type** `Number` and ensure that the **Required** checkbox is unchecked.

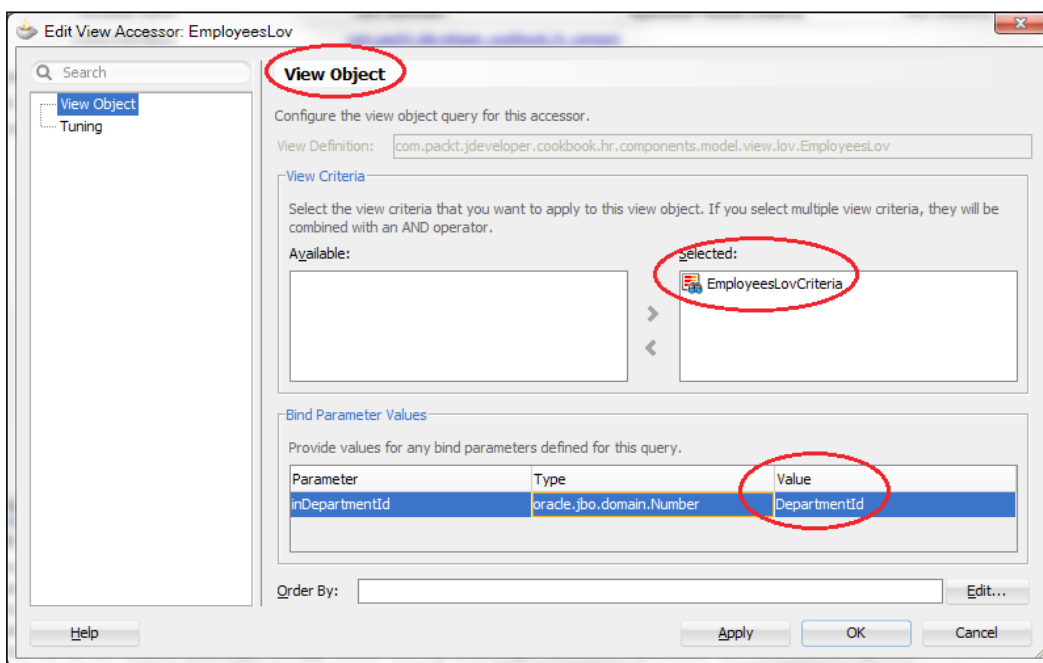


7. While in the **Query** section, click on the **Create new view criteria** button (the green plus sign icon) to add view criteria to the EmployeesLov view object. Click the **Add Item** button and select DepartmentId for the **Attribute**. For the **Operator** select **Equals** and for the **Operand** select **Bind Variable**. Ensure that you select the inDepartmentId bind variable that you created in the previous step from the **Parameter** combo. Make sure the **Ignore Null Values** checkbox is unchecked and that the **Validation** selection is **Optional**.



8. Back to the CascadingLovs view object, go to the **Attributes** section and select the DepartmentId attribute.
9. Click on the **List of Values** tab and then on the **Add list of values** button (the green plus sign icon).
10. On the **Create List of Values** dialog, click on the **Create new view accessor** button (the green plus sign icon) next to the **List Data Source** combo and add the DepartmentsLov view accessor.
11. Select **DepartmentId** for the **List Attribute**.
12. While on the **Create List of Values** dialog, click on the **UI Hints** tab and in the **Display Attributes** section shuttle the DepartmentName attribute from the **Available** list to the **Selected** list. Click **OK**.

13. Repeat steps 8 through 12 to add an LOV for the `EmployeeId` attribute. Use the `EmployeesLov` as the list data source and `EmployeeId` as the list attribute. For the display attributes in the **UI Hints** tab, use the `FirstName` and `LastName` attributes.
14. In the `CascadingLovs` view object, go to the **View Accessors** section select the `EmployeesLov` view accessor (do not click on the **View Definition** link). Then click on the **Edit selected View Accessor** button (the pen icon).
15. In the **Edit View Accessor** dialog, select the **View Object** section and shuttle the `EmployeesLovCriteria` from the **Available** list to the **Selected** list. Also, for the `inDepartmentId` parameter in the **Bind Parameter Values** section, enter `DepartmentId` in the **Value** field and click **OK**.



16. Double-click on the `HrComponentsAppModule` application module in the **Application Navigator** to open the application module definition.
17. Go to the **Data Model** section, select the `CascadingLovs` view object and shuttle it from the **Available View Objects** list to the **Data Model**.

How it works...

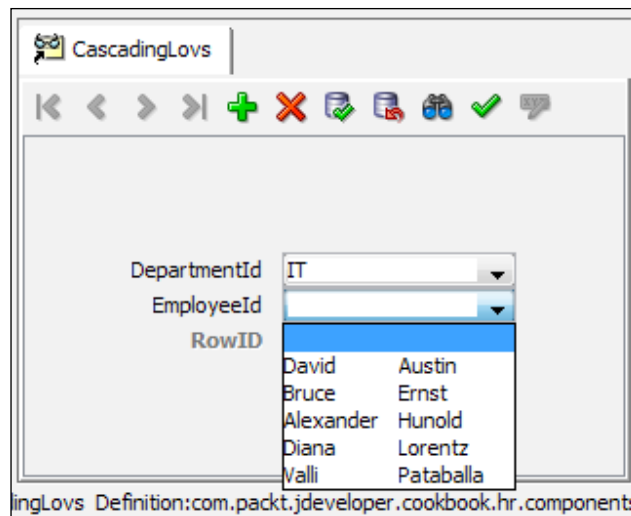
To demonstrate this recipe, we created a new table in the HR schema called `CASCADING_LOVS`. This table has two columns an `EMPLOYEE_ID` and a `DEPARTMENT_ID`. The table does not have a primary key constraint, so we will be able to freely add records to it. In a real world development project, a proper database design would require that all of your database tables have a primary key defined. Based on this table, we created an entity object called `CascadingLovs` (step 1). Since we did not indicate a primary key for the `CASCADING_LOVS` table, the framework asked us to indicate a primary key attribute (step 2). We did so by creating a key attribute called `RowID` based on the row's `ROWID`. Then we proceeded to create a view object called `CascadingLovs` based on the `CascadingLovs` entity object (step 3).

In order to setup LOVs for the `DepartmentId` and `EmployeeId` attributes, we had to create the LOV accessor view objects, namely the `DepartmentsLov` and the `EmployeesLov` view objects (steps 4 and 5). We also added named view criteria to the `EmployeesLov` (steps 6 and 7) based on the `inDepartmentId` bind variable. This way we will be able to control the result set produced by the `EmployeesLov` based on the department ID value. In step 7 when we created the view criteria, we saw that JDeveloper suggested a default name in the **Criteria Name** field. This is the name that is used to programmatically access the view criteria in your Java code. The name of the view criteria can be changed; however, we have chosen to use the default `EmployeesLovCriteria` provided by JDeveloper.

In steps 8 through 13, we proceeded by LOV-enabling the `DepartmentId` and `EmployeeId` attributes.

The important glue work was done in steps 14 and 15. In these steps, we edited the `EmployeesLov` view accessor and declaratively applied the `EmployeesLovCriteria` on the accessor. We also provided a value for the `inDepartmentId` bind variable using the expression `DepartmentId`, which indicates the value of the `DepartmentId` attribute at runtime. This is the `CascadingLovs` department identifier attribute that is updated by the controlling `DepartmentsLov` LOV. By doing so, we have set the controlling variable's value, the `inDepartmentId` bind variable, using the value provided by the `DepartmentsLov` data source, that is, the `DepartmentId`.

Finally, in steps 16 and 17, we added the `CascadingLovs` view object to the application module's data model, so that we may be able to test it using the ADF Model Tester. While running the ADF Model Tester, notice how the employees list is controlled by the selected department.



There's more...

For the cascading LOVs to work properly on the frontend Fusion web application user interface, you need to make sure that the `autoSubmit` property is set to `true` for the controlling LOV UI component. This will ensure that, upon selection, the controlling attribute's value is submitted to the server. The UI component's `autoSubmit` property can also be set to a default value by setting the attribute's **Auto Submit** property at the business component level. This can be done in the view object's **Attributes | UI Hints** tab.

Also, note the behavior of the controlled LOV based on the view criteria **Ignore Null Values** setting. When this checkbox is selected, null values for the criteria item will be ignored and the result set will not be filtered yielding all possible employee rows. In this case, the `EmployeesLov` view object's `WHERE` clause is amended by adding `OR (:inDepartmentId is null)` to the query. If the **Ignore Null Values** checkbox is not selected, then null values for the criteria item are not ignored, yielding no employees rows.

See also

- ▶ *Overriding `remove()` to delete associated children entities, Chapter 2, Dealing with Basics: Entity Objects*

Creating static LOVs

A static LOV is produced by basing its list data source view accessor on a static view object, that is, a view object that uses a static list as its data source. A static list is a list of constant data that you either enter manually or import from a text file using JDeveloper. A static list could also be produced by basing the view object on a query that generates static data. The advantage of using a static LOV is that you can display static read-only data in your application's user interface without having to create a database table for it. In all cases, the amount of static data presented to the user should be small.

In this recipe, we will create a static view object called `ColorLov` and use it as an LOV list data source to LOV-enable a transient attribute of the `Employees` view object.

Getting ready

This recipe was developed using the `HRComponents` workspace, which was created in the *Overriding `remove()` to delete associated children entities* recipe in *Chapter 2, Dealing with Basics: Entity Objects*. The `HRComponents` workspace requires a database connection to the HR schema.

How to do it...

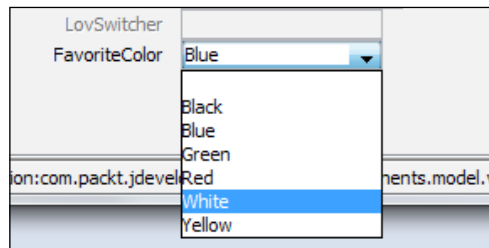
1. Create a new view object using the **Create View Object** wizard.
2. In the **Name** page, enter `ColorsLov` for the name of the view object and select **Static list** for the **Data Source**.
3. In the **Attributes** page, click on the **New...** button. Create an attribute called `ColorDesc`.
4. In the **Attribute Settings** page, select **Key Attribute** for the `ColorDesc` attribute.
5. In the **Static List** page, use the **Add Row** button (the green plus sign icon) to add the following data: `Black`, `Blue`, `Green`, `Red`, `White`, and `Yellow`.
6. Click **Finish** to complete the creation of the `ColorsLov` view object.
7. Add a new transient attribute to the `Employees` view object called `FavoriteColor`.
8. In the **Attributes | Details** tab for the `FavoriteColor` attribute, ensure that the **Updatable** property is set to **Always**.
9. Click on the **List of Values** tab and add an LOV called `LOV_FavoriteColor`. For the LOV **List Data Source**, use the **Create new view accessor** button (the green plus sign icon) and select the `ColorsLov` static view object.
10. For the LOV **List Attribute**, select the `ColorDesc` attribute.

How it works...

In steps 1 through 6, we went through the process of creating a view object, called `ColorsLov`, which uses a static list as its data source. We have indicated that the view object has one attribute called `ColorDesc` (step 3) and we have indicated that attribute as a key attribute (step 4). Notice in step 5 how the **Create View Object** wizard allows you to manually enter the static data. In the same **Static List** page, the wizard allows you to import data from a file in a comma-separated-values (CSV) format.

In order to test the static LOV, we added a transient variable called `FavoriteColor` to the `Employees` view object (step 7-8) and we LOV-enabled the attribute using the `ColorsLov` as the list data source view accessor (steps 9-10).

When we test the application module with the ADF Model Tester, the `FavoriteColor` attribute is indeed populated by the static values we have entered for the `ColorsLov` view object.



There's more...

Notice that the static data that is entered for the static view object is saved on a resource bundle. This allows you to localize the data as needed.

Also note that in some cases where localization of static data is not needed, a read-only view object that is based on a query producing static data can simulate a static view object. For instance, consider the read-only view object that is based on the following query:

```
SELECT 'Black' AS COLOR_DESC FROM DUAL
UNION
SELECT 'Blue' AS COLOR_DESC FROM DUAL
UNION
SELECT 'Green' AS COLOR_DESC FROM DUAL
UNION
SELECT 'Red' AS COLOR_DESC FROM DUAL
UNION
SELECT 'White' AS COLOR_DESC FROM DUAL
UNION
SELECT 'Yellow' AS COLOR_DESC FROM DUAL;
```

It can be used as a list data source for the `FavoriteColor` LOV producing the same results.

See also

- ▶ *Overriding remove() to delete associated children entities, Chapter 2, Dealing with Basics: Entity Objects*

Overriding bindParametersForCollection() to set a view object bind variable

There are times when you need to programmatically set the value of a bind variable used in the view object query. One way to accomplish this task is by overriding the view object method `bindParametersForCollection()` and explicitly specifying the value for the particular bind variable. This technique comes handy when the bind variable values cannot be specified in a declarative way, or the bind variable value source changes dynamically at runtime.

This recipe will show how to provide a default value for a bind variable used in the view object query if a value has not already been specified for it.

Getting ready

You will need to have access to the shared components workspace that was developed in the *Breaking up the application in multiple workspaces* recipe in *Chapter 1, Pre-requisites to Success: ADF Project Setup and Foundations*. Additional functionality will be added to the `ExtViewObjectImpl` and `ExtApplicationModuleImpl` custom framework classes that were developed in the *Setting up BC base classes* recipe in *Chapter 1, Pre-requisites to Success: ADF Project Setup and Foundations*.

This recipe is also using the `HRComponents` workspace, which was created in the *Overriding remove() to delete associated children entities* recipe in *Chapter 2, Dealing with Basics: Entity Objects*. The `HRComponents` workspace requires a database connection to the HR schema.

Moreover, we will modify the `EmployeeCount` view object, which was introduced in the *Using a method validator based on a view accessor* recipe in *Chapter 2, Dealing with Basics: Entity Objects*.

How to do it...

1. Open the shared components workspace.
2. Open the `ExtViewObjectImpl` view object framework extension class and add the following method to it:

```
protected void setBindVariableValue(Object[] bindVariables,  
    String name, Object value) {  
    // iterate all bind variables
```

```

    for (Object bindVariable : bindVariables) {
        // check for the specific bind variable name
        if (((Object[])bindVariable)[0].toString().equals(name)) {
            // set the bind variable's new value
            ((Object[])bindVariable)[1] = value;
            return;
        }
    }
}

```

2. Open the `ExtApplicationModuleImpl` application module framework extension class and add the following method to it:

```

public Object getCustomData(String key) {
    // base class returns no custom data
    return null;
}

```

3. Deploy the shared components workspace into an ADF Library JAR.
4. Open the `HRComponents` workspace.
5. Open the `HrComponentsAppModuleImpl` custom application module implementation class and override the `getCustomData()` method. In it, replace the `return super.getCustomData()` with the following:

```

return DEFAULT_DEPARTMENT_ID_KEY.equals(key)
    ? DEFAULT_DEPARTMENT_ID : null;

```

6. Open the `EmployeeCount` view object, go to the **Java** page and create a custom view object class.
7. Open the `EmployeeCountImpl` custom view object implementation class and override the `bindParametersForCollection()` method. Add the following code to it before the call to `super.bindParametersForCollection()`:

```

// if bind variable value has not been provided,
// provide a default setting
if (this.getDepartmentId() == null) {
    // get default department id
    Number departmentId = ((Number)((ExtApplicationModuleImpl)
        this.getApplicationModule()).getCustomData(
        DEFAULT_DEPARTMENT_ID_KEY));
    // set bind variable right on the query to
    // default variable
    super.setBindVariableValue(object, "DepartmentId",
        departmentId.toString());
    // set bind variable on view object as well,
    //to be available for this time forward
    this.setDepartmentId(departmentId);
}

```

8. Finally, add the `EmployeeCount` view object to the `HrComponentsAppModule` application module data model.

How it works...

In the recipe *Using a method validator based on a view accessor* in Chapter 2, *Dealing with Basics: Entity Objects*, we created a view object called `EmployeeCount`, which we used in order to get the employee count for specific departments. The view object was based on the following query:

```
SELECT COUNT(*) AS EMPLOYEE_COUNT
FROM EMPLOYEES
WHERE DEPARTMENT_ID = :DepartmentId
```

The `EmployeeCount` view object was added as a view accessor to the employee entity object and it was used in a method validator. In that method validator, a value was supplied programmatically for the `DepartmentId` bind variable prior to executing the `EmployeeCount` query.

In this recipe, we have used the same `EmployeeCount` view object to demonstrate how to supply a default value for the `DepartmentId` bind variable. We did this by creating a custom view object implementation class (step 7) and then by overriding its `bindParametersForCollection()` method (step 8). This method is called by the ADF-BC framework to allow you to set values for the view object query's bind variables. When the framework calls `bindParametersForCollection()`, it supplies among the other parameters an `Object []`, which contains the query's bind variables (the `bindVariables` parameter). We set a default value of the `DepartmentId` bind variable by calling `super.setBindVariableValue()`. This is the helper method that we added to the view object framework extension class in step 2. In the `setBindVariableValue()` method we iterate over the query's bind variables until we find the one we are looking for and once we find it, we set its new value.

Note that in `bindParametersForCollection()` we have called `getApplicationModule()` to get the application module for this `EmployeeCount` view object instance (having added the `EmployeeCount` view object to the `HrComponentsAppModule` data model in step 9). This method returns an `oracle.jbo.ApplicationModule` interface, which we cast to an `ExtApplicationModuleImpl`. As a recommended practice, you should not be accessing specific application modules from within your view objects. In this case, we relaxed the rule a bit by casting the `oracle.jbo.ApplicationModule` interface returned by the `getApplicationModule()` method to our application module framework extension class. We have then called `getCustomData()`, which we overrode in step 6, to get the default `DepartmentId` value. It is this default value (stored in variable `departmentId`) that we supply when calling `super.setBindVariableValue()`.

There's more...

Although the `executeQueryForCollection()` method of `ViewObjectImpl` method can be used to set the view object's query bind variable values, do not use this method because the framework will never invoke it when `getEstimatedRowCount()` is called to identify the result set's row count. If you do, `getEstimatedRowCount()` will not produce the correct row count as you are altering the query by supplying values to the query's bind variables.

Also, note that with the 11.1.1.5.0 (PS4) release, a new `ViewObjectImpl` method called `prepareRowSetForQuery()` was introduced that can be used to set the query's bind parameter values. The following code illustrates how to use it to set a value for the `DepartmentId` bind variable in this recipe:

```
public void prepareRowSetForQuery(ViewRowSetImpl vrsImpl) {
    // get default departmentId value as before
    Number departmentId =
        vrsImpl.ensureVariableManager().setVariableValue(
            "DepartmentId", departmentId);
    super.prepareRowSetForQuery(vrsImpl);
}
```

Both `bindParametersForCollection()` and `prepareRowSetForQuery()` are valid choices for setting the view object's query bind variable values. If for some reason both of them are overridden, note that the framework will first call `prepareRowSetForQuery()` and then `bindParametersForCollection()`.

See also

- ▶ *Setting up BC base classes, Chapter 1, Pre-requisites to Success: ADF Project Setup and Foundations*
- ▶ *Overriding `remove()` to delete associated children entities, Chapter 2, Dealing with Basics: Entity Objects*
- ▶ *Using a method validator based on a view accessor, Chapter 2, Dealing with Basics: Entity Objects*

Creating view criteria programmatically

View criteria augment the view object's `WHERE` clause by appending additional query conditions to it. They work in conjunction with the `af:query` ADF Faces UI component to provide query-by-example support to the frontend user interface. View criteria can be created declaratively in JDeveloper in the **Query** section of the view object definition by clicking on the **Create new view criteria** button (the green plus sign icon) in the **View Criteria** section. Programmatically, the ADF-BC API supports the manipulation of view criteria among others via the `ViewCriteria`, `ViewCriteriaRow`, and `ViewCriteriaItem` classes, and through a number of methods implemented in the `ViewObjectImpl` class. This technique comes handy when the view criteria cannot be specified during the design stage. One example might be the creation of a custom query-by-example page for your application, in which case the view criteria must be created programmatically at runtime.

In this recipe, we will see how to create view criteria programmatically. The use case will be to dynamically amend the `Employees` view object query by adding it to the view criteria. The values that we will use for the view criteria items will be obtained from the result set of yet another view object.

Getting ready

You will need to have access to the shared components workspace that was developed in the *Breaking up the application in multiple workspaces* recipe in *Chapter 1, Pre-requisites to Success: ADF Project Setup and Foundations*. The functionality will be added to the `ExtViewObjectImpl` custom framework class that was developed in the *Setting up BC base classes* recipe in *Chapter 1, Pre-requisites to Success: ADF Project Setup and Foundations*. For testing purposes, we will be using the `HRComponents` workspace, which was created in the *Overriding remove() to delete associated children entities* recipe in *Chapter 2, Dealing with Basics: Entity Objects*. The `HRComponents` workspace requires a database connection to the `HR` schema.

How to do it...

1. Open the shared components workspace.
2. Open the `ExtViewObjectImpl.java` view object framework extension class in the Java editor and add the following `searchUsingAdditionalCriteria()` method:

```
public void searchUsingAdditionalCriteria(
    ViewObject providerViewObject,
    String[] attribNames) {
    // create the criteria
    ViewCriteria vc = this.createViewCriteria();
    // set the view criteria name
    vc.setName("searchUsingAdditionalCriteria");
}
```

```

// AND with previous criteria
vc.setConjunction(ViewCriteriaComponent.VC_CONJ_AND);
// get criteria item data from the provider
// view object
RowSetIterator it =
    providerViewObject.createRowSetIterator(null);
it.reset();
while (it.hasNext()) {
    Row providerRow = it.next();
    // add a criteria item for each attribute
    for (String attribName : attribNames) {
        try {
            // create the criteria item
            ViewCriteriaRow vcRow = vc.createViewCriteriaRow();
            // set the criteria item value
            vcRow.setAttribute(attribName,
                providerRow.getAttribute(attribName));
            // add criteria item to the view criteria
            vc.insertRow(vcRow);
        } catch (JboException e) {
            LOGGER.severe(e);
        }
    }
}
// done with iterating provider view object
it.closeRowSetIterator();
// apply the criteria to this view object
this.applyViewCriteria(vc);
// execute the view object's query
this.executeQuery();
}

```

3. For logging purposes, add an `ADFLogger` to the same class as shown in the following code:

```

private static ADFLogger LOGGER = ADFLogger.createADFLogger(
    ExtViewObjectImpl.class);

```

4. Deploy the shared components projects into an ADF Library JAR.

5. For testing purposes, open the `HRCComponents` workspace and add the following `searchEmployeesUsingAdditionalCriteria()` method to the `HrComponentsAppModuleImpl` custom application module implementation class:

```
public void searchEmployeesUsingAdditionalCriteria() {
    // invoke searchUsingAdditionalCriteria() to create
    // result set based on View criteria item
    // data obtained from another view object's rowset
    this.getEmployees()
        .searchUsingAdditionalCriteria(this.getCascadingLoves(),
        new String[] { "EmployeeId" });
}
```

How it works...

In step 1, we created a method called `searchUsingAdditionalCriteria()` in the shared components workspace `ExtViewObjectImpl` view object framework extension class, to allow view objects to alter their queries by dynamically creating and applying view criteria. The data for the view criteria items are provided by another view object. The method accepts the view object that will provide the view criteria item data values (`providerViewObject`) and an array of attribute names (`attribNames`) that is used to create the view criteria items, and also retrieve the data from the provider view object. The following lines show how this method is called from the `searchEmployeesUsingAdditionalCriteria()` method that we added to the application module implementation class in step 5:

```
((ExtViewObjectImpl) this.getEmployees())
    .searchUsingAdditionalCriteria(this.getCascadingLoves(),
    new String[] { "EmployeeId" });
```

As you can see, we have used the view object returned by `this.getCascadingLoves()` in order to obtain the view criteria item values. A single criteria item based on the employee ID was also used.

In the `searchUsingAdditionalCriteria()`, we first called `createViewCriteria()` to create view criteria for the view object. This returns an `oracle.jbo.ViewCriteria` object representing the view criteria. This object can be used subsequently to add criteria items onto it. Then we called `setConjunction()` on the view criteria to set the conjunction operator (OR, AND, UNION, NOT). The conjunction operator is used to combine multiple criteria when nested view criteria are used by the view object. This could be the case if the view object has defined additional view criteria. We have used an AND conjunction in this example (the `ViewCriteriaComponent.VC_CONJ_AND` constant), although this can easily be changed by passing the conjunction as another parameter to `searchUsingAdditionalCriteria()`.

In order to retrieve the view criteria item data, we iterated the provider view object, and for each row of data we called `createViewCriteriaRow()` on the view criteria to create the criteria row. This method returns an `oracle.jbo.ViewCriteriaRow` object representing a criteria row. We added the view criteria row data by calling `setAttribute()` on the newly created criteria row, and we added the criteria row to the view criteria by calling `insertRow()` on the view criteria, passing the criteria row as an argument.

Once all criteria items have been setup, we call `applyViewCriteria()` on the view object, specifying the newly created view criteria. Then we call `executeQuery()` to execute the view object's query based on the applied view criteria. The result set produced matches the applied criteria.

There's more...

Note what happens when the framework executes the view object query after applying the view criteria programmatically. Adding two criteria rows, for example, will append the following to the query's `WHERE` clause:

```
( ( (Employee.EMPLOYEE_ID = :vc_temp_1 ) ) OR ( (Employee.EMPLOYEE_ID
= :vc_temp_2 ) ) )
```

As you can see, the framework amends the query using temporary bind variables (`vc_temp_1`, `vc_temp_2`, and so on) for each criteria row.

Also note the following:

- ▶ Calling `applyViewCriteria()` on the view object erases any previously applied criteria. In order to preserve these, the framework provides another version of `applyViewCriteria()` that accepts an extra `bAppend` Boolean parameter. Based on the value of `bAppend`, the newly applied criteria can be appended to the existing criteria, if any. Moreover, to apply multiple criteria at once, the framework provides the `setApplyViewCriteriaNames()` method. This method accepts a `java.lang.String` array of the criteria names to apply, and by default `ANDs` the criteria applied.
- ▶ The way the `setAttribute()` method of `ViewCriteriaRow` was used in this recipe sets up an equality operation for the criterion, that is, `EmployeeId = someValue`. In order to specify a different operation for the criterion item, you must specify the operation as part of the `setAttribute()` method call. For example, `vcRow.setAttribute("EmployeeId", "< 150")`, `vcRow.setAttribute("EmployeeId", "IN (100,200,201)")` and so on.

- ▶ Finally, note that you can setup the view criteria item via the `setOperator()` and `setValue()` methods supplied by the `ViewCriteriaItem` class. You will need to call `ensureCriteriaItem()` on the criteria row in order to get access to a `ViewCriteriaItem`. The following is an example:

```
// get the criteria item from the criteria row
ViewCriteriaItem criteriaItem =
    vcRow.ensureCriteriaItem("EmployeeId");
// set the criteria item operator
criteriaItem.setOperator("<");
// set the criteria item value
criteriaItem.getValues().get(0).setValue(new Integer(150));
```

See also

- ▶ *Setting up BC base classes, Chapter 1, Pre-requisites to Success: ADF Project Setup and Foundations*
- ▶ *Overriding remove() to delete associated children entities, Chapter 2, Dealing with Basics: Entity Objects*

Clearing the values of bind variables associated with the view criteria

This recipe shows you how to clear the values associated with bind variables used as operands in view criteria items for a specific view object. It implements a method called `clearCriteriaVariableValues()` in the view object framework extension class, which becomes available for all view objects to call. Bind variables are associated as operands for criteria items during the process of creating the view object's view criteria in the **Create View Criteria** dialog. Also, as we have seen in the *Creating view criteria programmatically* recipe, bind variables are generated automatically by the framework when programmatically creating view criteria. You can use this technique when you want to clear the search criteria on a search form based on some user action. A use case might be, for instance, that you want to immediately clear the search criteria after the search button is pressed.

Getting ready

You will need to have access to the shared components workspace that was developed in the *Breaking up the application in multiple workspaces* recipe in *Chapter 1, Pre-requisites to Success: ADF Project Setup and Foundations*. The functionality will be added to the `ExtViewObjectImpl` custom framework class that was developed in the *Setting up BC base classes* recipe in *Chapter 1, Pre-requisites to Success: ADF Project Setup and Foundations*.

How to do it...

1. Open the shared components workspace.
2. Open the `ExtViewObjectImpl.java` view object framework extension class in the Java editor and add the following `clearCriteriaValues()` method:

```
public void clearCriteriaVariableValues(
    String[] criteriaNames) {
    // iterate all view criteria names
    for (String criteriaName : criteriaNames) {
        // get the view criteria
        ViewCriteria vc = this.getViewCriteria(criteriaName);
        if (vc != null) {
            VariableValueManager vvm = vc.ensureVariableManager();
            Variable[] variables = vvm.getVariables();
            for (Variable var: variables) {
                vvm.setVariableValue(var, null);
            }
        }
    }
}
```

How it works...

The `clearCriteriaVariableValues()` is added to the `ExtViewObjectImpl` view object framework extension class, thus making it available for all view objects to call it. The method accepts a `java.lang.String` array of view criteria names (`criteriaNames`) and iterates over them, getting the associated view criteria for each of them. It then calls `ensureVariableManager()` on the view criteria to retrieve the bind variables manager, an `oracle.jbo.VariableValueManager` interface, which is implemented by the framework to manage named variables.

The bind variables are retrieved by calling `getVariables()` on the variable manager. This method returns an array of objects implementing the `oracle.jbo.Variable` interface, the actual bind variables. Finally, we iterate over the bind variables used by the view criteria, setting their values to null by calling `setVariableValue()` for each one of them.

There's more...

Note that the technique used in the recipe does not remove the view criteria associated with the view object, it simply resets the values of the bind variables associated with the view criteria. In order to completely remove the view criteria associated with a particular view object, call the `ViewObjectImpl` method `removeViewCriteria()`. This method first unapplies the specific view criteria and then completely removes them from the view object. If you want to unapply the view criteria without removing them from the view object, use the `removeApplyViewCriteriaName()` method. Furthermore, you can also clear all the view object view criteria in effect by calling `applyViewCriteria()` on the view object and specifying `null` for the view criteria name. Finally, to clear any view criteria in effect, you can also delete all the view criteria rows from it using the `remove()` method. Any of the above calls will alter the view criteria for the lifetime of the specific view object instance until the next time any of these calls are invoked again.

See also

- ▶ *Setting up BC base classes, Chapter 1, Pre-requisites to Success: ADF Project Setup and Foundations*

Searching case insensitively using view criteria

This recipe shows you a technique that you can use to handle case-insensitive (or case sensitive for that matter) searching for strings, when using view criteria for a view object. The framework provides various methods, such as `setUpperColumns()` and `isUpperColumns()`, for instance, at various view criteria levels (`ViewCriteria`, `ViewCriteriaRow` and `ViewCriteriaItem`) that can be used to construct generic helper methods to handle case searching. This technique can be used to allow case-insensitive or case-sensitive searches in your application based on some controlling user interface component or some application configuration option. For instance, a custom search form can be constructed with a checkbox component indicating whether the search will be case sensitive or not.

Getting ready

You will need to have access to the shared components workspace that was developed in the *Breaking up the application in multiple workspaces* recipe in *Chapter 1, Pre-requisites to Success: ADF Project Setup and Foundations*. The functionality will be added to the `ExtViewObjectImpl` custom framework class that was developed in the *Setting up BC base classes* recipe in *Chapter 1, Pre-requisites to Success: ADF Project Setup and Foundations*.

How to do it...

1. Open the shared components workspace.
2. Open the `ExtViewObjectImpl.java` view object framework extension class in the Java editor and add the following `setViewCriteriaCaseInsensitive()` method:

```
public void setViewCriteriaCaseInsensitive(
    boolean bCaseInsensitive) {
    // get all View Criteria managed by this view object
    ViewCriteria[] vcList = getAllViewCriterias();
    if (vcList != null) {
        // iterate over all view criteria
        for (ViewCriteria vc : vcList) {
            // set case-insensitive or case-sensitive as
            // indicated by the bCaseInsensitive parameter
            if (vc.isUpperColumns() != bCaseInsensitive)
                vc.setUpperColumns(bCaseInsensitive);
        }
    }
}
```

How it works...

We have added the `setViewCriteriaCaseInsensitive()` method in the `ExtViewObjectImpl` view object framework extension class to allow all view objects to call it in order to enable or disable case-insensitive search based on view criteria managed by the specific view object. The boolean parameter `bCaseInsensitive` indicates whether case-insensitive search is to be enabled for the view criteria.

The method gets access to all view criteria managed by the specific view object by calling `getAllViewCriterias()`. This framework method returns an `oracle.jbo.ViewCriteria` array containing all view criteria (both applied and unapplied) that are managed by the view object. It then iterates over them, checking in each iteration whether the current case-insensitive setting, obtained by calling `isUpperColumns()`, differs from the desired setting indicated by `bCaseInsensitive`. If this is the case, case-insensitivity is set (or reset) by calling `setUpperColumns()` for the specific view criteria.

When you enable case-insensitive search for the view criteria, the framework—when it adjusts the view object a query, based on the view criteria—calls the `UPPER()` database function in the `WHERE` clause for those criteria items where case-insensitive search has been enabled. This behavior can be seen when you declaratively define view criteria using the **Create View Criteria** dialog. Notice how the **View Object Where Clause** is altered as you check and uncheck the **Ignore Case** checkbox. This behavior is achieved programmatically as explained in this recipe by calling `setUpperColumns()`.

There's more...

As mentioned earlier, the framework allows you to control case-insensitive search at various levels. In this recipe, we have seen how to affect case searching for the view criteria as a whole, by utilizing the `setUpperColumns()` method defined for the `ViewCriteria` object. Individual criteria rows and items can be set separately by calling `setUpperColumns()` for specific `ViewCriteriaRow` and `ViewCriteriaItem` objects respectively.

See also

- ▶ *Setting up BC base classes, Chapter 1, Pre-requisites to Success: ADF Project Setup and Foundations*

5

Putting them all together: Application Modules

In this chapter, we will cover:

- ▶ Creating and using generic extension interfaces
- ▶ Exposing a custom method as a web service
- ▶ Accessing a service interface method from another application module
- ▶ A passivation/activation framework for custom session-specific data
- ▶ Displaying application module pool statistics
- ▶ Using a shared application module for static lookup data
- ▶ Using a custom database transaction

Introduction

An application module in the ADF Business Components framework represents a basic transactional unit that implements specific business use cases. It encompasses a data model comprising a hierarchy of view objects and optionally other application module instances, along with a number of custom methods that together implement a specific business use case. It allows the creation of **bindings** at the ViewController project layer, through the corresponding application model **data control** and the **ADF model layer (ADFm)**. Moreover, it allows for the creation of custom functionality that can be exposed through its client interface and subsequently bound as method bindings. Method bindings declaratively *bind* user interface components to back-end data and services providing data access.

Custom application module methods can easily be exposed as web services through the application module service interface. Moreover, application modules and their configured view object instances can be exposed as service data object (SDO) components for consumption in a SOA infrastructure.

Creating and using generic extension interfaces

Back in *Chapter 1, Pre-requisites to Success: ADF Project Setup and Foundations* in the *Setting up BC base classes* recipe, we introduced a number of framework extension classes for various business components. We did this so that we could provide common implementation functionality for all derived business components throughout the application. In this recipe, we will go over how to expose parts of that common functionality as a generic extension interface. By doing so, this generic interface becomes available to all derived business components, which in turn can expose it to their own client interface and make it available to the ViewController layer through the bindings layer.

Getting ready

You will need to have access to the `SharedComponents` workspace that was developed in the *Breaking up the application in multiple workspaces* recipe in *Chapter 1, Pre-requisites to Success: ADF Project Setup and Foundations*. Additional functionality will be added to the `ExtApplicationModuleImpl` custom framework class that was developed in the *Setting up BC base classes* recipe in *Chapter 1, Pre-requisites to Success: ADF Project Setup and Foundations*.

This recipe also uses the `HRComponents` workspace, which was created in the *Overriding remove() to delete associated children entities* recipe in *Chapter 2, Dealing with Basics: Entity Objects*. The `HRComponents` workspace requires a database connection to the HR schema.

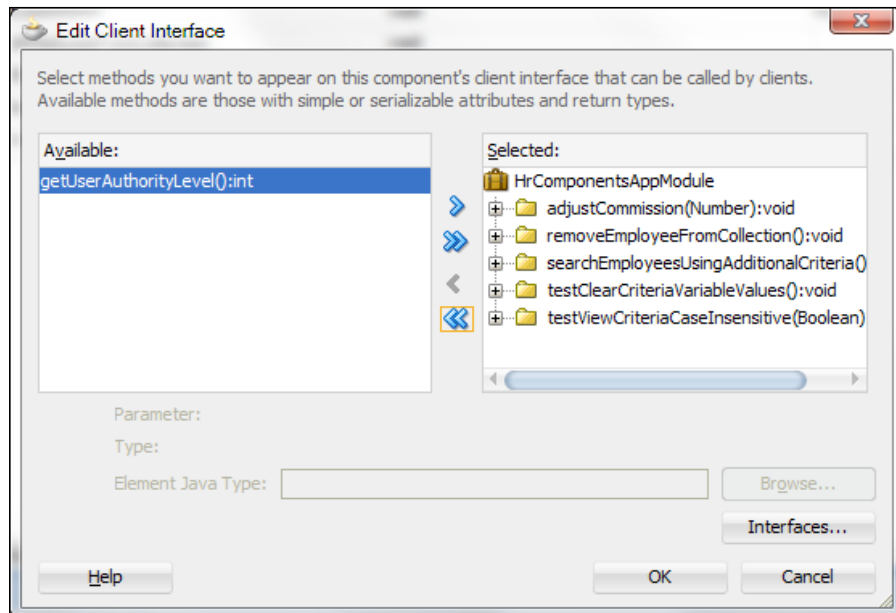
How to do it...

1. Open the shared components workspace in JDeveloper.
2. Create an interface called `ExtApplicationModule` as follows:

```
public interface ExtApplicationModule {
    // return some user authority level, based on
    // the user's name
    public int getUserAuthorityLevel();
}
```

3. Locate and open the custom application module framework extension class `ExtApplicationModuleImpl`. Modify it so that it implements the `ExtApplicationModule` interface.
4. Then, add the following method to it:


```
public int getUserAuthorityLevel() {
    // return some user authority level, based on the user's name
    return ("anonymous".equalsIgnoreCase(this.
getUserPrincipalName()))?
        AUTHORITY_LEVEL_MINIMAL : AUTHORITY_LEVEL_NORMAL;
}
```
5. Rebuild the `SharedComponents` workspace and deploy it as an ADF Library JAR.
6. Now, open the `HrComponents` workspace.
7. Locate and open the `HrComponentsAppModule` application module definition.
8. Go to the **Java** section and click on the **Edit application module client interface** button (the pen icon in the **Client Interface** section).
9. On the **Edit Client Interface** dialog, shuttle the `getUserAuthorityLevel()` interface from the **Available** to the **Selected** list.



How it works...

In steps 1 and 2, we have opened the `SharedComponents` workspace and created an interface called `HrComponentsAppModule`. This interface contains a single method called `getUserAuthorityLevel()`.

Then, we updated the application module framework extension class `HrComponentsAppModuleImpl` so that it implements the `HrComponentsAppModule` interface (step 3). We also implemented the method `getUserAuthorityLevel()` required by the interface (step 4). For the sake of this recipe, this method returns a user authority level based on the authenticated user's name. We retrieve the authenticated user's name by calling `getUserPrincipal().getName()` on the `SecurityContext`, which we retrieve from the current ADF context (`ADFContext.getCurrent().getSecurityContext()`). If security is not enabled for the ADF application, the user's name defaults to `anonymous`. In this example, we return `AUTHORITY_LEVEL_MINIMAL` for anonymous users, and for all others we return `AUTHORITY_LEVEL_NORMAL`. We rebuilt and redeployed the `SharedComponents` workspace in step 5.

In steps 6 through 9, we opened the `HRComponents` workspace and added the `getUserAuthorityLevel()` method to the `HrComponentsAppModuleImpl` client interface. By doing this, we exposed the `getUserAuthorityLevel()` generic extension interface to a derived application module, while keeping its implementation in the base framework extension class `ExtApplicationModuleImpl`.

There's more...

Note that the steps followed in this recipe to expose an application module framework extension class method to a derived class' client interface can be followed for other business components framework extension classes as well.

See also

- ▶ *Setting up BC base classes, Chapter 1, Pre-requisites to Success: ADF Project Setup and Foundations*
- ▶ *Overriding `remove()` to delete associated children entities, Chapter 2, Dealing with Basics: Entity Objects*

Exposing a custom method as a web service

Service-enabling an application module allows you, among others, to expose custom application module methods as web services. This is one way for service consumers to consume the service-enabled application module. The other possibilities are accessing the application module by another application module, and accessing it through a Service Component Architecture (SCA) composite. Service-enabling an application module allows access to the same application module both through web service clients and interactive web user interfaces. In this recipe, we will go over the steps involved in service-enabling an application module by exposing a custom application module method to its service interface.

Getting ready

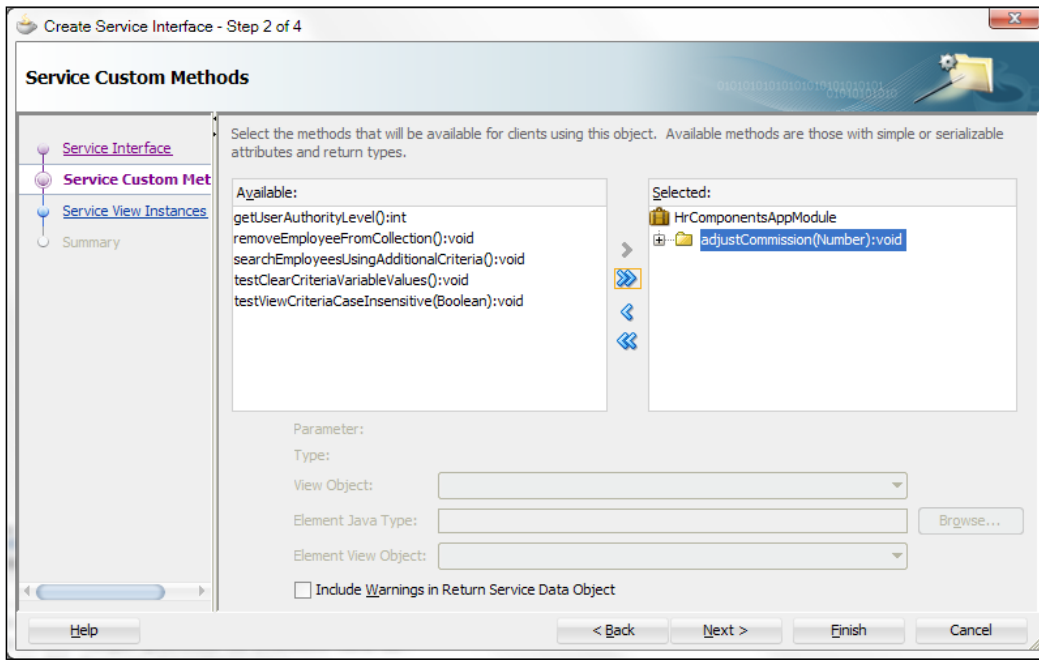
This recipe was developed using the `HRComponents` workspace, which was created in the *Overriding `remove()` to delete associated children entities* recipe in *Chapter 2, Dealing with Basics: Entity Objects*. The `HRComponents` workspace requires a database connection to the HR schema.

Furthermore, for this recipe, we will expose the `adjustCommission()` application module method that was developed back in *Chapter 3, A Different Point of View: View Objects Techniques* for the *Iterating a view object using a secondary rowset iterator* recipe as a web service.

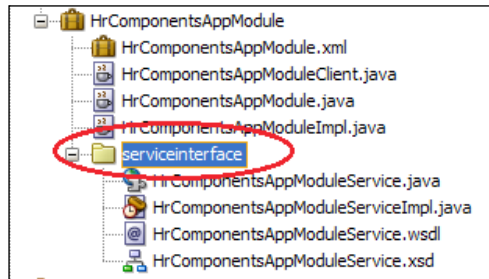
How to do it...

1. Open the `HRComponents` project in JDeveloper.
2. Double-click on the `HRComponentsAppModule` application module in the **Application Navigator** to open its definition.
3. Go to the **Service Interface** section and click on the **Enable support for Service Interface** button (the green plus sign icon in the **Service Interface** section). This will start the **Create Service Interface** wizard.
4. In the **Service Interface** page, accept the defaults and click **Next**.

5. In the **Service Custom Methods** page, locate the `adjustCommission()` method and shuttle it from the **Available** list to the **Selected** list. Click on **Finish**.

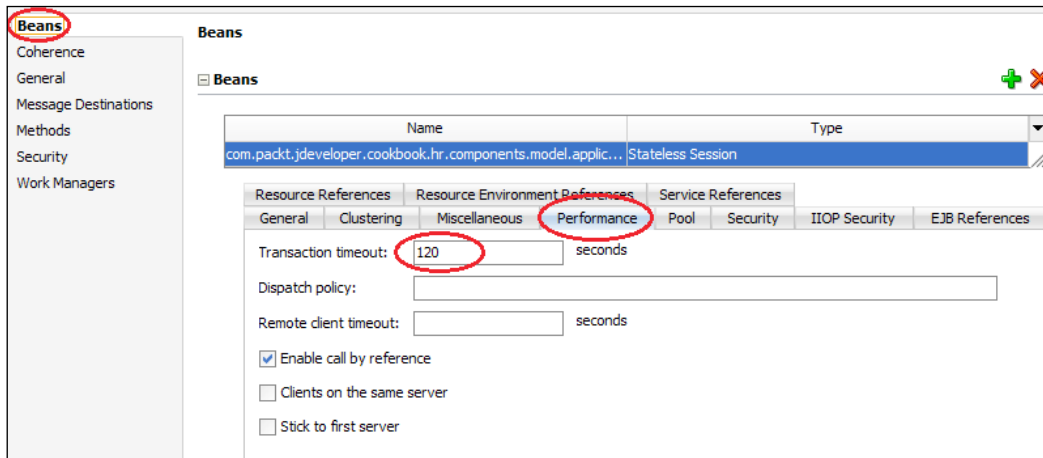


6. Observe that the `adjustCommission()` method is shown in the **Service Interface Custom Methods** section of the application module's **Service Interface**. The service interface files were generated in the `serviceinterface` package under the application module and are shown in the **Application Navigator**.



7. Double-click on the `weblogic-ejb-jar.xml` file under the `META-INF` package in the **Application Navigator** to open it.

8. In the **Beans** section, select the `com.packt.jdeveloper.cookbook.hr.components.model.application.common.HrComponentsAppModuleServiceBean` bean and click on the **Performance** tab. For the Transaction timeout field, enter 120.



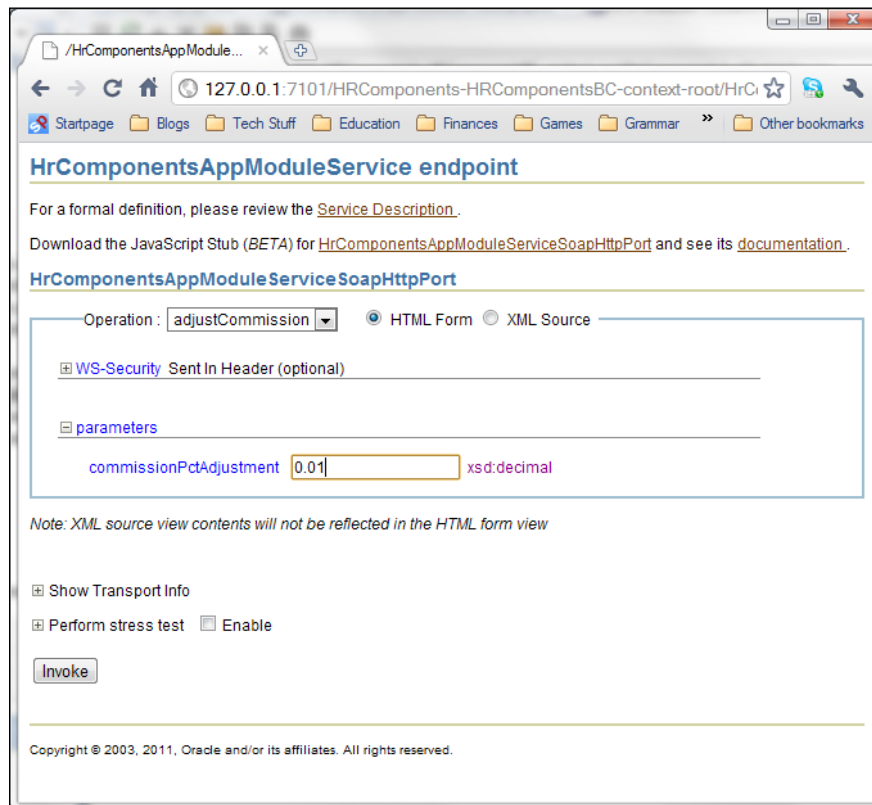
How it works...

In steps 1 through 6, we have exposed the `adjustCommission()` custom application module method to the application module's service interface. This is a custom method that adjusts all the Sales department employees' commissions by the percentage specified. As a result of exposing the `adjustCommission()` method to the application module service interface, JDeveloper generates the following files:

- ▶ `HrComponentsAppModuleService.java`: Defines the service interface
- ▶ `HrComponentsAppModuleServiceImpl.java`: The service implementation class
- ▶ `HrComponentsAppModuleService.xsd`: The service schema file describing the input and output parameters of the service
- ▶ `HrComponentsAppModuleService.wsdl`: The Web Service Definition Language (WSDL) file, describing the web service
- ▶ `ejb-jar.xml`: The EJB deployment descriptor. It is located in the `src/META-INF` directory
- ▶ `weblogic-ejb-jar.xml`: The WebLogic-specific EJB deployment descriptor, located in the `src/META-INF` directory

In steps 7 and 8, we adjust the service Java Transaction API (JTA) transaction timeout to 120 seconds (the default is 30 seconds). This will avoid any exceptions related to transaction timeouts when invoking the service. This is an optional step added specifically for this recipe, as the process of adjusting the commission for all sales employees might take longer than the default 30 seconds, causing the transaction to time out.

To test the service using the JDeveloper integrated WebLogic application server, right-click on the `HrComponentsAppModuleServiceImpl.java` service implementation file in the **Application Navigator** and select **Run** or **Debug** from the context menu. This will build and deploy the `HrComponentsAppModuleService` web service into the integrated WebLogic server. Once the deployment process is completed successfully, you can click on the service URL in the **Log** window to test the service. This will open a test window in JDeveloper and also enable the HTTP Analyzer. Otherwise, copy the target service URL from the **Log** window and paste it into your browser's address field. This will bring up the service's endpoint page.



On this page, select the **adjustCommission** method from the **Operation** drop down, specify the **commissionPctAdjustment** parameter amount and click on the **Invoke** button to execute the web service. Observe how the employees' commissions are adjusted in the `EMPLOYEES` table in the `HR` schema.

There's more...

For more information on service-enabling application modules consult chapter *Integrating Service-Enabled Application Modules* in the *Fusion Developer's Guide for Oracle Application Development Framework* which can be found at http://docs.oracle.com/cd/E24382_01/web.1112/e16182/toc.htm.

See also

- ▶ *Overriding remove() to delete associated children entities, Chapter 2, Dealing with Basics: Entity Objects*
- ▶ *Iterating a view object using a secondary rowset iterator, Chapter 3, A Different Point of View: View Objects Techniques*

Accessing a service interface method from another application module

In the recipe *Exposing a custom method as a web service* in this chapter, we went through the steps required to service-enable an application module and expose a custom application module method as a web service. We will continue in this recipe by explaining how to invoke the custom application module method, exposed as a web service, from another application module.

Getting ready

This recipe will call the `adjustCommission()` custom application module method that was exposed as a web service in the *Exposing a custom method as a web service* recipe in this chapter. It requires that the web service is deployed in WebLogic and that it is accessible.

The recipe also requires that both the `SharedComponents` workspace and the `HRComponents` workspace are deployed as ADF Library JARs and that are added to the workspace used by this specific recipe. Additionally, a database connection to the `HR` schema is required.

How to do it...

1. Ensure that you have built and deployed both the `SharedComponents` and `HRComponents` workspaces as ADF Library JARs.
2. Create a File System connection in the **Resource Palette** to the directory path where the `SharedComponents.jar` and `HRComponents.jar` ADF Library JARs are located. In the book source code, they are located in the `chapter5/recipe3/ReusableJARs` directory.

3. Create a new **Fusion Web Application (ADF)** called `HRComponentsCaller` using the **Create Fusion Web Application (ADF)** wizard.
4. Create a new application module called `HRComponentsCallerAppModule` using the **Create Application Module** wizard. In the **Java** page, check on the **Generate Application Module Class** checkbox to generate a custom application module implementation class. JDeveloper will ask you for a database connection during this step, so make sure that a new database connection to the `HR` schema is created.
5. Expand the **File System | ReUsableJARs** connection in the **Resource Palette** and add both the `SharedComponents` and `HRComponents` libraries to the project. You do this by right-clicking on the jar file and selecting **Add to Project...** from the context menu.
6. Bring up the business components **Project Properties** dialog and go to the **Libraries and Classpath** section. Click on the **Add Library...** button and add the **BC4J Service Client** and **JAX-WS Client** extensions.
7. Double-click on the `HRComponentsCallerAppModuleImpl.java` custom application module implementation file in the **Application Navigator** to open it in the Java editor.

8. Add the following method to it:

```
public void adjustCommission(  
    BigDecimal commissionPctAdjustment) {  
    // get the service proxy  
    HrComponentsAppModuleService service =  
        (HrComponentsAppModuleService)ServiceFactory  
            .getServiceProxy(  
                HrComponentsAppModuleService.NAME);  
    // call the adjustCommission() service  
    service.adjustCommission(commissionPctAdjustment);  
}
```

9. Expose `adjustCommission()` to the `HRComponentsCallerAppModule` client interface.
10. Finally, in order to be able to test the `HRComponentsCallerAppModule` application module with the ADF Model Tester, locate the `connections.xml` file in the **Application Resources** section of the **Application Navigator** under the **Descriptors | ADF META-INF** node, and add the following configuration to it:

```
<Reference  
    name="{/com/packt/jdeveloper/cookbook/hr/components/model/  
    application/common/}HrComponentsAppModuleService"  
    className="oracle.jbo.client.svc.Service" xmlns="">  
<Factory  
    className="oracle.jbo.client.svc.ServiceFactory"/>  
<RefAddresses>  
<StringRefAddr addrType="serviceInterfaceName">
```

```

<Contents>com.packt.jdeveloper.cookbook.hr.components.model.
application.common.serviceinterface.HrComponentsAppModuleService
</Contents>
</StringRefAddr>
<StringRefAddr addrType="serviceEndpointProvider">
<Contents>ADFBC</Contents>
</StringRefAddr>
<StringRefAddr addrType="jndiName">
<Contents>HrComponentsAppModuleServiceBean#com.packt.jdeveloper.
cookbook.hr.components.model.application.common.
serviceinterface.HrComponentsAppModuleService</Contents>
</StringRefAddr>
<StringRefAddr addrType="serviceSchemaName">
<Contents>HrComponentsAppModuleService.xsd</Contents>
</StringRefAddr>
<StringRefAddr addrType="serviceSchemaLocation">
<Contents>com/packt/jdeveloper/cookbook/hr/components/model/
application/common/serviceinterface/</Contents>
</StringRefAddr>
<StringRefAddr addrType="jndiFactoryInitial">
<Contents>weblogic.jndi.WLInitialContextFactory</Contents>
</StringRefAddr>
<StringRefAddr addrType="jndiProviderURL">
<Contents>t3://localhost:7101</Contents>
</StringRefAddr>
</RefAddresses>
</Reference>

```

How it works...

In steps 1 and 2, we have made sure that both the `SharedComponents` and `HRComponents` ADF Library JARs are deployed and that a file system connection was created, in order that both of these libraries get added to a newly created project (in step 5). Then, in steps 3 and 4, we create a new Fusion web application based on ADF, and an application module called `HRComponentsCallerAppModule`. It is from this application module that we intend to call the `adjustCommission()` custom application module method, exposed as a web service by the `HrComponentsAppModule` service-enabled application module in the `HRComponents` library JAR. For this reason, in step 4, we have generated a custom application module implementation class. We proceed by adding the necessary libraries to the new project in steps 5 and 6. Specifically, the following libraries were added: `SharedComponents.jar`, `HRComponents.jar`, `BC4J Service Client`, and `JAX-WS Client`.

In steps 7 through 9, we create a custom application module method called `adjustCommission()`, in which we write the necessary glue code to call our web service. In it, we first retrieve the web service proxy, as a `HrComponentsAppModuleService` interface, by calling `ServiceFactory.getServiceProxy()` and specifying the name of the web service, which is indicated by the constant `HrComponentsAppModuleService.NAME` in the service interface. Then we call the web service through the retrieved interface.

In the last step, we have provided the necessary configuration in the `connections.xml` so that we will be able to call the web service from an RMI client (the ADF Model Tester). This file is used by the web service client to locate the web service. For the most part, the `<Reference>` information that was added to it was generated automatically by JDeveloper in the *Exposing a custom method as a Web service* recipe, so it was copied from there. The extra configuration information that had to be added is the necessary JNDI context properties `jndiFactoryInitial` and `jndiProviderURL` that are needed to resolve the web service on the deployed server. You should change these appropriately for your deployment. Note that these parameters are the same as the initial context parameters used to lookup the service when running in a managed environment.

To test calling the web service, ensure that you have first deployed it and that it is running. You can then use the ADF Model Tester, select the **adjustCommission** method and execute it.

There's more...

For additional information related to such topics as securing the ADF web service, enabling support for binary attachments, deploying to WebLogic, and more, refer to the Integrating Service-Enabled Application Modules section in the Fusion Developer's Guide for Oracle Application Development Framework which can be found at http://docs.oracle.com/cd/E24382_01/web.1112/e16182/toc.htm.

See also

- ▶ *Setting up BC base classes, Chapter 1, Pre-requisites to Success: ADF Project Setup and Foundations*
- ▶ *Overriding `remove()` to delete associated children entities, Chapter 2, Dealing with Basics: Entity Objects*
- ▶ *Exposing a custom method as a web service, Chapter 5, Putting them all together: Application Modules*

A passivation/activation framework for custom session-specific data

In order to improve performance and preserve a stateful notion while utilizing a stateless protocol (that is, HTTP) the ADF Business Components framework implements the concept of **application module pooling**. This is a technique of maintaining a limited number of application modules, the exact number specified by configuration, in a pool, which are preserved across multiple user requests for the same HTTP session. If an application module instance for a session already exists in the application module pool, it gets reused. When all available application modules in the pool have been associated with specific sessions, an application module already linked with a particular session must be freed. This requires that the data associated with the application module is saved.

The process of saving the information associated with the specific application module is called **passivation**. The information is stored in a **passivation store**, usually a database, in XML format. The opposite process of restoring the state of the application module from the passivation store is called **activation**. Custom data is associated with specific application modules, and therefore with specific user sessions, by using a `Hashtable` obtained from an `oracle.jbo.Session` object. The `Hashtable` is obtained by calling `getSession().getUserData()` from the application module implementation class.

If you are using such custom data as part of some algorithm in your application and you expect the custom data to persist from one user request to another, passivation (and subsequent activation) support for these custom data must be implemented programmatically. You can add custom passivation and activation logic to your application module implementation class by overriding the `ApplicationModuleImpl` methods `passivateState()` and `activateState()` respectively. The `passivateState()` method creates the necessary XML elements for the application module's custom data that must be passivated. Conversely, the `activateState()` method detects the specific XML elements that identify the custom data in the passivated XML document and restores them back into the session custom data.

This recipe will show you how to do this, and at the same time build a mini framework to avoid duplication of the basic passivation/activation code that you must write for all the application modules in your project.

Getting ready

You will need to have access to the `SharedComponents` workspace that was developed in the *Breaking up the application in multiple workspaces* recipe in *Chapter 1, Pre-requisites to Success: ADF Project Setup and Foundations*. Additional functionality will be added to the `ExtApplicationModuleImpl` custom framework class that was developed in the *Setting up BC base classes* recipe in *Chapter 1, Pre-requisites to Success: ADF Project Setup and Foundations*.

This recipe is also using the `HRCComponents` workspace, which was created in the *Overriding `remove()` to delete associated children entities* recipe in *Chapter 2, Dealing with Basics: Entity Objects*. The `HRCComponents` workspace requires a database connection to the HR schema.

How to do it...

1. Open the `SharedComponents` workspace in JDeveloper and load the `ExtApplicationModuleImpl` application module framework extension class in the Java editor.
2. Add the following methods to the `ExtApplicationModuleImpl` application module framework extension class:

```
protected String[] onStartPassivation() {
    // default implementation: no passivation ids
    // are defined
    return new String[] { };
}
protected String onPassivate(String passivationId) {
    // default implementation: passivates nothing
    return null;
}
protected void onEndPassivation() {
    // default implementation: does nothing
}
protected String[] onStartActivation() {
    // default implementation: no activation ids
    // are defined
    return new String[] { };
}
protected void onActivate(String activationId,
    String activationData) {
    // default implementation: activates nothing
}
protected void onEndActivation() {
    // default implementation: does nothing
}
```

3. Override the void `passivateState(Document, Element)` method. Add the following code after the call to `super.passivateState()`:

```
// begin custom data passivation: returns a
// list of the custom data passivation identifiers
String[] passivationIds = onStartPassivation();
// process all passivation identifiers
for (String passivationId : passivationIds) {
    // check for valid identifier
```

```

if (passivationId != null &&
    passivationId.trim().length() > 0) {
    // passivate custom data: returns
    // the passivation data
    String passivationValue =
        onPassivate(passivationId);
    // check for valid passivation data
    if (passivationValue != null &&
        passivationValue.length() > 0) {
        // create a new text node in the
        // passivation XML
        Node node =
            document.createElement(passivationId);
        Node cNode =
            document.createTextNode(passivationValue);
        node.appendChild(cNode);
        // add the passivation node to the
        // parent element
        element.appendChild(node);
    }
}
// inform end of custom data passivation
onEndPassivation();

```

4. Override the `activateState(Element element)` method. Add the following code after the call to `super.activateState()`:

```

// check for element to activate
if (element != null) {
    // begin custom data activation: returns a
    // list of the custom data activation identifiers
    String[] activationIds = onStartActivation();
    // process all activation identifiers
    for (String activationId : activationIds) {
        // check for valid identifier
        if (activationId != null &&
            activationId.trim().length() > 0) {
            // get nodes from XML for the specific
            // activation identifier
            NodeList nl =
                element.getElementsByTagName(activationId);
            // if it was found in the activation data
            if (nl != null) {
                // activate each node
            }
        }
    }
}

```



```
        for (int n = 0, length =
            nl.getLength(); n < length; n++) {
            Node child =
                nl.item(n).getFirstChild();
            if (child != null) {
                // do the actual custom data
                // activation
                onActivate(activationId,
                    child.getNodeValue().toString());
                break;
            }
        }
    }
}
// inform end of custom data activation
onEndActivation();
}
```

5. Rebuild and redeploy the SharedComponents ADF Library JAR.
6. Open the HRComponents workspace and load the HrComponentsAppModuleImpl and HrComponentsAppModule application module custom implementation classes into the Java editor.
7. Add the following getActivationPassivationIds() helper method. Also, ensure that you define a constant called CUSTOM_DATA_PASSIVATION_ID indicating the custom data passivation identifier.

```
private static final String CUSTOM_DATA_PASSIVATION_ID =
    "customDataPassivationId";
private String[] getActivationPassivationIds() {
    // return the passivation/activation identifiers
    return new String[] { CUSTOM_DATA_PASSIVATION_ID };
}
```

8. Override the onStartPassivation(), onPassivate(), onStartActivation(), and onActivate() methods. Provide the following implementation for them:

```
protected String[] onStartPassivation() {
    // return the passivation identifiers
    return getActivationPassivationIds();
}
protected String onPassivate(String passivationId) {
    String passivationData = null;
    // passivate this application module's
    // custom data only
}
```

```

    if (CUSTOM_DATA_PASSIVATION_ID.equals(
passivationId)) {
        // return the custom data from the Application
        // Module session user data
        passivationData = (String)getSession()
            .getUserData().get(CUSTOM_DATA_PASSIVATION_ID);
    }
    return passivationData;
}
protected String[] onStartActivation() {
    // return the activation identifiers
    return getActivationPassivationIds();
}
protected void onActivate(String activationId,
String activationData) {
    // activate this application module's custom data only
    if (CUSTOM_DATA_PASSIVATION_ID.equals(activationId)) {
        // add custom data to the Application
        // Module's session
        getSession().getUserData().put(
            CUSTOM_DATA_PASSIVATION_ID, activationData);
    }
}
}

```

9. Finally, for testing purposes, override the `prepareSession()` method and add the following code after the call to `super.prepareSession()`:

```

// add some custom data to the Application
// Module session
getSession().getUserData()
    .put(CUSTOM_DATA_PASSIVATION_ID,
        "Some custom data");


```

How it works...

In the first two steps, we have laid out a basic passivation/activation framework by adding a number of methods to the `ExtApplicationModuleImpl` application module framework extension class dealing specifically with this process. Specifically, these methods are:

- ▶ `onStartPassivation()`: The framework calls this method to indicate that a passivation process is about to start. Derived application modules that need to passivate custom data will override this method and return a `java.lang.String` array of passivation identifiers, indicating custom data that needs to be passivated.

- ▶ `onPassivate()`: The framework calls this method to indicate that some specific custom data, identified by the `passivationId` parameter, needs to be passivated. Derived application modules will override this method to passivate the specific custom data. It returns the passivated data as a `java.lang.String`.
- ▶ `onEndPassivation()`: This method is called by the framework to indicate that the passivation process is complete. Derived application modules could override this method to perform post-passivation actions.
- ▶ `onStartActivation()`: This method is called by the framework to indicate that an activation process is about to begin. Derived application modules in need of activating custom data, should override this method and return a list of activation identifiers.
- ▶ `onActivate()`: This method is called by the framework when some custom data—that is, the parameter `activationData`—needs to be activated. The custom data is identified by a unique identifier indicated by the parameter `activationId`. Derived application modules should override this method and restore the custom data being activated into the application module's user data `Hashtable`.
- ▶ `onEndActivation()`: This method indicates the end of the activation process. It can be overridden by derived application modules to do some post-activation actions.


 These methods do nothing at the base class level. It is when they are overridden by derived application modules (see step 8) that they come to life.

In step 3, we have overridden the ADF Business Components framework method `passivateState()` and hooked up our own passivation/activation framework to it. ADF calls this method to indicate that a passivation is taking place. In it, after calling `super.passivateState()` to allow for the ADF processing, we first call `onStartPassivation()`. If a derived application module has overridden this method, it should return a list of passivation identifiers. These identifiers should uniquely identify the application module custom data that needs to be passivated at the application module level. We then iterate over the passivation identifiers, calling `onPassivate()` each time to retrieve the passivation data. We create a new XML node for the passivation identifier, we add the passivation data to it and append it to the parent XML node that is passed as a parameter by the ADF framework (the `element` parameter) to `passivateState()`. When all passivation identifiers have been processed, `onEndPassivation()` is called.

Step 4 is somewhat similar and does the activation. In this case, we have overridden the ADF `activateState()` method, which is called by the framework to indicate that the activation process is taking place. In it, we first call `super.activateState()` to allow for framework processing and then call `onStartActivation()` to get a list of the activation identifiers. We iterate over the activation identifiers, looking for each identifier in the activated XML data for the application module element. This is done by calling `element.getElementsByTagName()`. This method could possibly return multiple nodes, so for each we call `onActivate()` to activate the specific custom data. When we call `onActivate()`, we pass the activation identifier and the activation data to it as arguments. It is then the responsibility of the derived application module to handle the specifics of the activation. Finally, when all activation identifiers have been processed, we call `onEndActivation()` to indicate that the activation process has ended.

After we have added these changes to the `ExtApplicationModuleImpl` application module framework extension class, we make sure that the `SharedComponents` ADF Library JAR was redeployed (in step 5).

In steps 6 through 8, we have added passivation/activation support for custom data to the `HrComponentsAppModule` application module in the `HRComponents` workspace. This is done by overriding the `onStartPassivation()`, `onPassivate()`, `onStartActivation()`, and `onActivate()` methods (in step 8). The list of passivation and activation identifiers comes from the `getActivationPassivationIds()` method that we added in step 7. For this recipe, only a single custom data, identified by the constant `CUSTOM_DATA_PASSIVATION_ID`, is passivated. Custom data is saved at the user data `Hashtable` in the `oracle.jbo.Session` associated with the specific application module. It is retrieved by calling `getSession().getUserData().get(CUSTOM_DATA_PASSIVATION_ID)` in the `onPassivate()` method. Similarly, it is set in `onActivate()` by calling `getSession().getUserData().put(CUSTOM_DATA_PASSIVATION_ID and activationData())`.

 In this case, the activation data is passed as an argument (the `activationData` parameter) to the `onActivate()` by the `activateState()` implemented in application module framework extension class, as in step 4.

Finally, note the code in step 9. In the overridden `prepareSession()`, we have initialized the custom data by calling `getSession().getUserData().put(CUSTOM_DATA_PASSIVATION_ID, "Some custom data")`.

To test the custom data passivation/activation framework, run the application module with the ADF Model Tester. The ADF Model Tester provides support for passivation and activation via the **Save Transaction State** and **Restore Transaction State** menu items under the **File** menu. Observe the generated passivation XML data in the JDeveloper Log window when **File | Save Transaction State** is chosen. In particular, observe that the `<customDataPassivationId>Some custom data</customDataPassivationId>` node is added to the `<AM>` node of the passivated XML document. This is the session data added in step 9 for testing purposes to demonstrate this passivation/activation framework.

```
[81] **syncSequenceIncrementSize** altered sequence 'increment by' value to 50
[82] <AM MonVer="0">
  <cd></cd>
  <TXN Def="0" New="0" Lok="2" tsi="0" pcid="1"/>
  <CONN/>
  <VO>
    <VO sig="1310500543716" qf="0" RS="0" Def="com.packt.jdeveloper.cookbook.hr.components.model.view.Employees" Name="Employees"/>
    <VO sig="1310500543836" qf="0" RS="0" Def="com.packt.jdeveloper.cookbook.hr.components.model.view.Departments"
Name="Departments"/>
    <VO sig="1310500543836" qf="1" RS="0" Def="com.packt.jdeveloper.cookbook.hr.components.model.view.Departments"
Name="DepartmentsManaged"/>
    <VO sig="1310500543836" qf="1" RS="0" Def="com.packt.jdeveloper.cookbook.hr.components.model.view.Employees"
Name="EmployeesManaged"/>
    <VO sig="1310500543836" qf="1" RS="0" Def="com.packt.jdeveloper.cookbook.hr.components.model.view.Employees"
Name="DepartmentEmployees"/>
  </VO>
  <customDataPassivationId>Some custom data</customDataPassivationId>
</AM>
```

There's more...

Note that the `activateState()` method is called by the ADF Business Components framework after the view objects instances associated with the application module have been activated by the framework. If you need to activate custom data that would be subsequently accessed by your view objects, then you will need to enhance the custom data passivation/activation framework by overriding `prepareForActivation()` and provide the activation logic there instead.

Also, note that the ADF Business Components framework provides similar `passivateState()` and `activateState()` methods at the view object level for passivating and activating view object custom data. In this case, custom data is stored in the user data `Hashtable` of the `oracle.jbo.Session` associated with the specific application module that contains the particular view object in its data model.

Finally, observe the following points:

- ▶ This framework does not cover the passivation/activation of view object custom data. If needed, you will need to expand this framework to support this extra requirement.
- ▶ It is important that during the development process you test your application modules for being activation-safe. This is done by disabling the application module pooling in the application module configuration. For more information on this topic, consult the *Testing to Ensure Your Application Module is Activation-Safe* section in the *Fusion Developer's Guide for Oracle Application Development Framework*.

See also

- ▶ *Setting up BC base classes, Chapter 1, Pre-requisites to Success: ADF Project Setup and Foundations*
- ▶ *Overriding remove() to delete associated children entities, Chapter 2, Dealing with Basics: Entity Objects*

Displaying application module pool statistics

In the *A passivation/activation framework for custom session-specific data* recipe in this chapter, we touched upon how application module pools are used by the ADF Business Components framework. In this recipe, we will introduce the `oracle.jbo.common.ampool.PoolMgr` application module pool manager and `oracle.jbo.common.ampool.ApplicationPool` application module pool classes, and explore how they can be utilized to collect statistical pool information. This may come in handy when debugging.

The use case that will be implemented by the recipe is to collect application module statistics and make them available in a generic view object, that can then be used by all application modules to gather and present statistical information to the frontend user interface.

Getting ready

You will need to have access to the `SharedComponents` workspace that was developed in the *Breaking up the application in multiple workspaces* recipe in *Chapter 1, Pre-requisites to Success: ADF Project Setup and Foundations*. Additional functionality will be added to the `ExtApplicationModuleImpl` custom framework class that was developed in the *Setting up BC base classes* recipe in *Chapter 1, Pre-requisites to Success: ADF Project Setup and Foundations*.

This recipe also uses the `HRComponents` workspace, which was created in the *Overriding remove() to delete associated children entities* recipe in *Chapter 2, Dealing with Basics: Entity Objects*. The `HRComponents` workspace requires a database connection to the HR schema.

How to do it...

1. Open the `SharedComponents` workspace in JDeveloper.
2. Create a new view object called `ApplicationModulePoolStatistics` using the following SQL query as its data source:

```
SELECT NULL AS POOL_NAME, NULL AS APPLICATION_MODULE_CLASS, NULL
AS AVAILABLE_INSTANCE_COUNT, NULL AS INIT_POOL_SIZE, NULL AS
INSTANCE_COUNT, NULL AS MAX_POOL_SIZE, NULL AS
```

```
NUM_OF_STATE_ACTIVATIONS, NULL AS NUM_OF_STATE_PASSIVATIONS,  
NULL AS NUM_OF_INSTANCES_REUSED, NULL AS REF_INSTANCES_RECYCLED,  
NULL AS UNREF_INSTANCES_RECYCLED, NULL AS  
REFERENCED_APPLICATION_MODULES, NULL AS NUM_OF_SESSIONS, NULL AS  
AVG_NUM_OF_SESSIONS_REF_STATE FROM DUAL
```

3. With the exception of the `PoolName` and `ApplicationModuleClass` attributes, which should be `String` data types, all other attributes should be `Number` types.
4. Designate the `PoolName` and `ApplicationModuleClass` attributes as key attributes.
5. In the **Java** section, create a custom view row class and ensure that the **Include accessors** checkbox is also checked.
6. Open the `ExtApplicationModuleImpl` application module custom framework class in the Java editor and add the following two methods to it:

```
public ExtViewObjectImpl  
    getApplicationModulePoolStatistics() {  
    return (ExtViewObjectImpl)findViewObject(  
        "ApplicationModulePoolStatistics");  
    }  
public void getAMPoolStatistics() {  
    // get the pool manager  
    PoolMgr poolMgr = PoolMgr.getInstance();  
    // get the pools managed  
    Enumeration keys = poolMgr.getResourcePoolKeys();  
    // iterate over pools  
    while (keys != null && keys.hasMoreElements()) {  
        // get pool name  
        String poolname = (String)keys.nextElement();  
        // get the pool  
        ApplicationPool pool =  
            (ApplicationPool)poolMgr.getResourcePool(poolname);  
        // get the pool statistics  
        Statistics statistics = pool.getStatistics();  
        // get and populate pool statistics view object  
        ExtViewObjectImpl amPoolStatistics =  
            getApplicationModulePoolStatistics();  
        if (amPoolStatistics != null) {  
            // empty the statistics  
            amPoolStatistics.executeEmptyRowSet();  
            // create and fill a new statistics row  
            ApplicationModulePoolStatisticsRowImpl poolInfo  
                (ApplicationModulePoolStatisticsRowImpl)  
                amPoolStatistics.createRow();  
            poolInfo.setPoolName(pool.getName());  
        }  
    }  
}
```

```

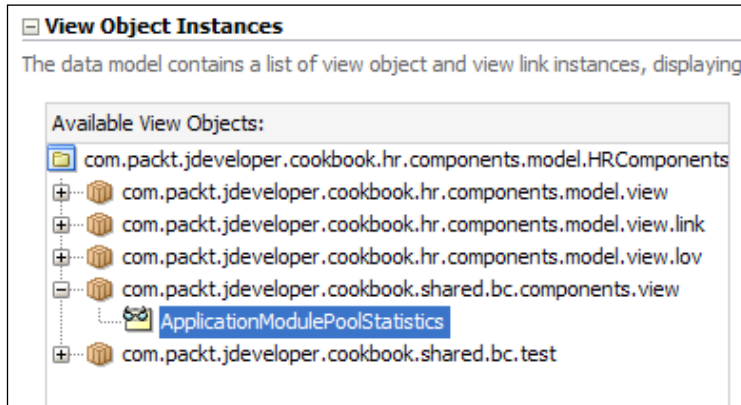
poolInfo.setApplicationModuleClass(
pool.getApplicationModuleClass());
poolInfo.setAvailableInstanceCount(new
    Number(pool.getAvailableInstanceCount()));
poolInfo.setInitPoolSize(new
    Number(pool.getInitPoolSize()));
poolInfo.setInstanceCount(new
    Number(pool.getInstanceCount()));
poolInfo.setMaxPoolSize(new
    Number(pool.getMaxPoolSize()));
poolInfo.setNumOfStateActivations(new
    Number(statistics.mNumOfStateActivations));
poolInfo.setNumOfStatePassivations(new
    Number(statistics.mNumOfStatePassivations));
poolInfo.setNumOfInstancesReused(new
    Number(statistics.mNumOfInstancesReused));
poolInfo.setRefInstancesRecycled(new
    Number(statistics.mNumOfReferencedInstancesRecycled));
poolInfo.setUnrefInstancesRecycled(new
    Number(statistics.mNumOfUnreferencedInstancesRecycled));
poolInfo.setReferencedApplicationModules(new
    Number(statistics.mReferencedApplicationModules));
poolInfo.setNumOfSessions(new
    Number(statistics.mNumOfSessions));
poolInfo.setAvgNumOfSessionsRefState(new
    Number(statistics.mAvgNumOfSessionsReferencingState));
// add the statistics
amPoolStatistics.insertRow(poolInfo);
    }
}
}

```

7. Open the `ExtApplicationModule` application module custom framework interface and add the following code to it:

```
public void getAMPoolStatistics();
```
8. Redeploy the `SharedComponents` ADF Library JAR.
9. Now, open the `HrComponents` workspace and in the **Resource Palette** create a file system connection for the `ReUsableJARs` directory where the `SharedComponents.jar` is deployed. Add the `SharedComponents.jar` to the `HrComponentsBC` business components project.
10. Double-click on the `HrComponentsAppModule` application module in the **Application Navigator** to open its definition.

11. Go to the **Data Model** section and locate the **ApplicationModulePoolStatistics** view object in the **Available View Objects** list. Shuttle it to the **Data Model** list.



12. Finally, go to the **Java** section, locate and add the `getAMPoolStatistics()` method to the `HRComponents` application module client interface.

How it works...

In steps 1 through 5, we created `ApplicationModulePoolStatistics`, a read-only view object, which we used to collect the application module pool statistics. By adding this view object to the `SharedComponents` workspace, it becomes available to all other projects in all workspaces throughout the ADF application that import the `SharedComponents` ADF Library JAR. In step 6, we have added the necessary functionality to collect the application module statistics and populate the `ApplicationModulePoolStatistics` view object. This is done in the `getAMPoolStatistics()` method. This gets an instance of the `oracle.jbo.common.ampool.PoolMgr` application module pool manager, via the call to the static `getInstance()`, along with an Enumeration of the application module pools managed by the pool manager by calling `getResourcePoolKeys()` on the pool manager. We iterate over all the pools managed by the manager and retrieve each pool using `getResourcePool()` on the pool manager. Then for each pool we call `getStatistics()` to get the pool statistics. We create a new `ApplicationModulePoolStatistics` view object row and populate it with the statistics information.

In step 7, we have added the `getAMPoolStatistics()` to the `ExtApplicationModule` application module framework extension interface, so that it becomes available to all application modules throughout the application.

In steps 8 and 9, we redeploy the `SharedComponents` library and created a file system connection in the **Resource Palette**. We use this file system connection to add the shared components `SharedComponents.jar` ADF Library JAR to the `HRComponents` business components project.

In steps 10 and 11, we add the `ApplicationModulePoolStatistics` view object to the `HrComponentsAppModule` application module data model. Notice how the `ApplicationModulePoolStatistics` view object is listed in the available view objects list, although it is implemented in the `SharedComponents` workspace.

Finally, in step 12, we add `getAMPoolStatistics()` to the `HrComponentsAppModule` application module client interface. By doing so, we will be able to call it using the ADF Model Tester.

To test the recipe, run the `HrComponentsAppModule` application module with the ADF Model Tester. In the ADF Model Tester double-click on the `HrComponentsAppModule` application module to open it, select the **getAMPoolStatistics** method from the **Method** combo, and click on the **Execute** button. Then open the **ApplicationModulePoolStatistics** view object to see the results.

Now you can bind both the `getAMPoolStatistics` method and the `ApplicationModulePoolStatistics` view object to any of your `ViewController` projects in your ADF application, and present a visual of this statistical information for debugging purposes.

There's more...

Note that the `oracle.jbo.common.ampool.ApplicationPool` interface provides a method called `dumpPoolStatistics()` to dump all pool statistics to a `PrintWriter` object. You can use this method to quickly print the application module pool statistics to the JDeveloper **Log** window, as shown in following code:

```
PrintWriter out = new PrintWriter(System.out, true);
pool.dumpPoolStatistics(new PrintWriter(out));
out.flush();
```

See also

- ▶ *Setting up BC base classes, Chapter 1, Pre-requisites to Success: ADF Project Setup and Foundations*
- ▶ *Overriding `remove()` to delete associated children entities, Chapter 2, Dealing with Basics: Entity Objects*
- ▶ *Creating and using generic extension interfaces, Chapter 5, Putting them all together: Application Modules*

Using a shared application module for static lookup data

Shared application modules allow you to share static read-only data models across multiple user sessions. They are the ideal place to collect all the static read-only view accessors used throughout your ADF application for validation purposes or as data sources for your list of values (LOVs). This is because a single shared application module is constructed and used throughout the ADF application for all user sessions, thus minimizing the system resources used by it. In this case, a single database connection is used. In addition, by collecting all of your static read-only view objects in a shared application module, you avoid possible duplication and redefinition of read-only view objects throughout your ADF application.

Internally, the ADF Business Components framework manages a pool of query collections for each view object as it is accessed by multiple sessions by utilizing a query collection pool, something comparable to application module pools used for session-specific application modules. The framework offers a number of configuration options to allow for better management of this pool. Moreover, as multiple threads will access the data, the framework partitions the iterator space by supporting multiple iterators for the same rowset, preventing race conditions among iterators on different sessions.

In this recipe, we will define a shared application module called `HrSharedAppModule`, and we will migrate to it all of the static read-only view objects defined for the `HrComponents` project. Furthermore, we will update all the view objects that currently reference these static read-only view objects, so that they are now referencing the view objects in the shared application module.

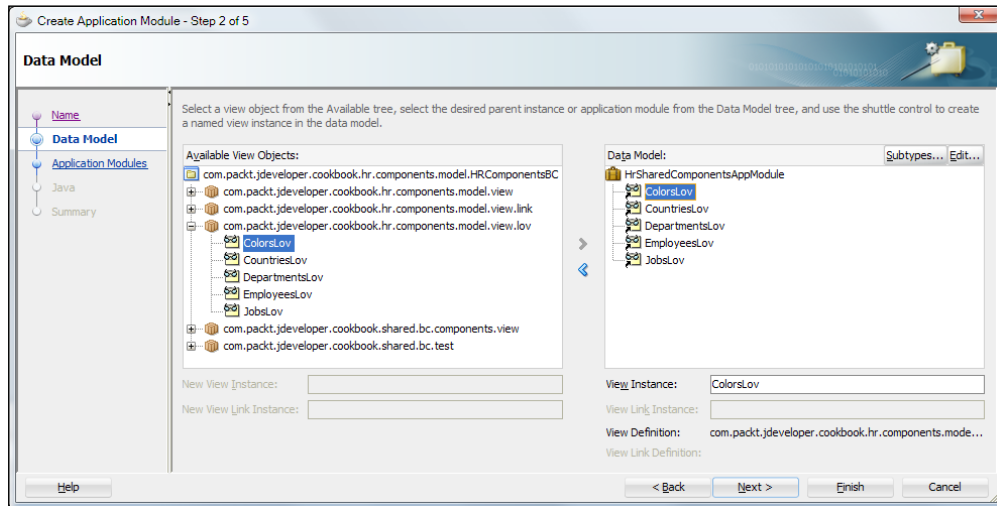
Getting ready

This recipe was developed using the `HRComponents` workspace, which was created in the *Overriding `remove()` to delete associated children entities* recipe in *Chapter 2, Dealing with Basics: Entity Objects*. The `HRComponents` workspace requires a database connection to the HR schema.

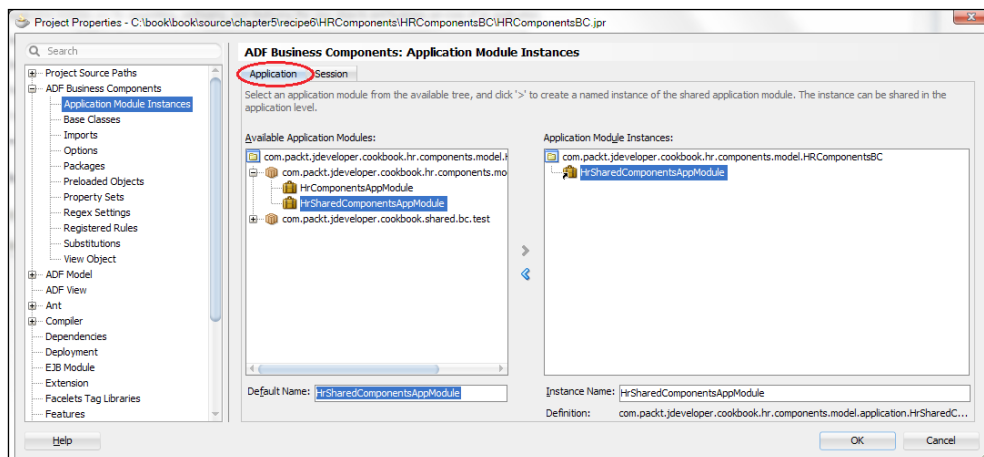
How to do it...

1. Right-click on the `com.packt.jdeveloper.cookbook.hr.components.model.application` package on the **Application Navigator** and select **New Application Module...**
2. Follow the steps in the **Create Application Module** wizard to create an application module called `HrSharedComponentsAppModule`.

- In the **Data Model** page, expand the `com.packt.jdeveloper.cookbook.hr.components.model.view.lov` package and shuttle all of the view objects currently under this package from the **Available View Objects** list to the **Data Model** list. Click on **Finish** when done.

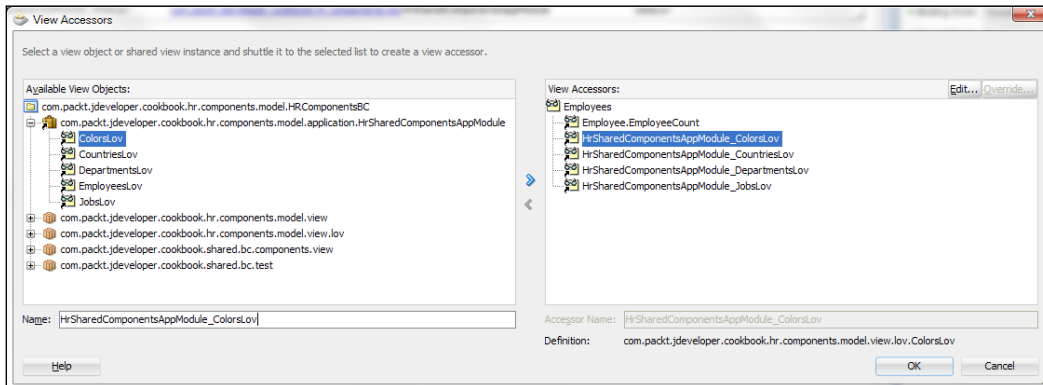


- Now, double-click on the `HRComponentsBC` in the **Application Navigator** to bring up the **Project Properties** dialog.
- Locate the **Application Module Instances** page by selecting **ADF Business Components | Application Module Instances** in the selection tree.
- Click on the **Application** tab and shuttle the `HrSharedComponentsAppModule` application module from the **Available Application Modules** list to the **Application Module Instances** list. Click **OK** to dismiss the **Project Properties** dialog.



Putting them all together: Application Modules

- For each view object that was added to the **HrSharedComponentsAppModule** shared application module, locate (through **Find Usages**) where it is used as a view accessor, and change its usage so that it is referenced from inside the **HrSharedComponentsAppModule** shared application module. The following screenshot shows the view accessors that were added to the **Employees** view object.



- Also for each view accessor used as a data source for an LOV, ensure that you are now using the view accessor included in **HrSharedComponentsAppModule**.

How it works...

In steps 1 through 6, we have defined a new application module called **HrSharedComponentsAppModule** and added all static read-only view objects developed so far—throughout the **HRComponents** business components project—in its data model. We have indicated that **HrSharedComponentsAppModule** will be a shared application module through the **Application Module Instances** page in the **Project Properties**, when we indicate that **HrSharedComponentsAppModule** will be an instance at application-level rather than an instance at session-level (in steps 4 through 6). By defining an application module at application-level, we allow all user sessions to access the same view instances contained in the application module data model.

In steps 7 and 8, we have identified all read-only view objects used as view accessors throughout the **HRComponents** business components project and updated each one at a time, so that the view object instance residing within the shared **HrSharedComponentsAppModule** application module is used. We have also ensured that for each LOV, we redefined its data source by using the updated view accessor.

There's more...

Ideally, the shared application module should contain a static read-only data model. If you expect that the data returned by any of the view objects might be updated, ensure that it will always return the latest data from the database by setting the view object **Auto Refresh** property to **true** in the **Tuning** section of the **Property Inspector**. This property is accessible while in the **General** section of the view object definition.

Page Iterator:	Full	▼	▼
Passivate State:	<default> (NoTransient)	▼	▼
Auto Refresh:	true	▼	▼
Access Mode:	<default> (DEFAULT)	▼	▼

The auto-refresh feature relies on the database change notification feature, so ensure that the data source database user has database notification privileges. This can be achieved by issuing the following `grant` command for the database connection user:

```
grant change notification to <ds_user_name>
```

For more information on shared application modules, consult the *Sharing Application Module View Instances* chapter in the *Fusion Developer's Guide for Oracle Application Development Framework* which can be found at http://docs.oracle.com/cd/E24382_01/web.1112/e16181/toc.htm.

See also

- ▶ *Overriding remove() to delete associated children entities, Chapter 2, Dealing with Basics: Entity Objects*

Using a custom database transaction

In the *Setting up BC base classes recipe in Chapter 1, Pre-requisites to Success: ADF Project Setup and Foundations*, we introduced a number of custom framework extension classes for most of the ADF business components. Among these are classes that can be used to extend the global ADF framework transaction implementation, in particular the `ExtDatabaseTransactionFactory` and `ExtDBTransactionImpl2` classes. In this recipe, we will cover how to use these classes, so that we can implement our own custom transaction implementation. The use case for this recipe will be to provide logging support for all transaction commit and rollback operations.

Getting ready

You will need to have access to the `SharedComponents` workspace that was developed in the *Breaking up the application in multiple workspaces* recipe in *Chapter 1, Pre-requisites to Success: ADF Project Setup and Foundations*. Additional functionality will be added to the `ExtDatabaseTransactionFactory` and `ExtDBTransactionImpl2` custom framework classes that were developed in the *Setting up BC base classes* recipe in *Chapter 1, Pre-requisites to Success: ADF Project Setup and Foundations*.

This recipe also uses the `HRComponents` workspace, which was created in the *Overriding `remove()` to delete associated children entities* recipe in *Chapter 2, Dealing with Basics: Entity Objects*. The `HRComponents` workspace requires a database connection to the `HR` schema.

How to do it...

1. Open the `SharedComponents` workspace and open the `ExtDatabaseTransactionFactory.java` file in the Java editor.
2. Override the `DatabaseTransactionFactory create()` method and replace the `return super.create()` method with the following code:

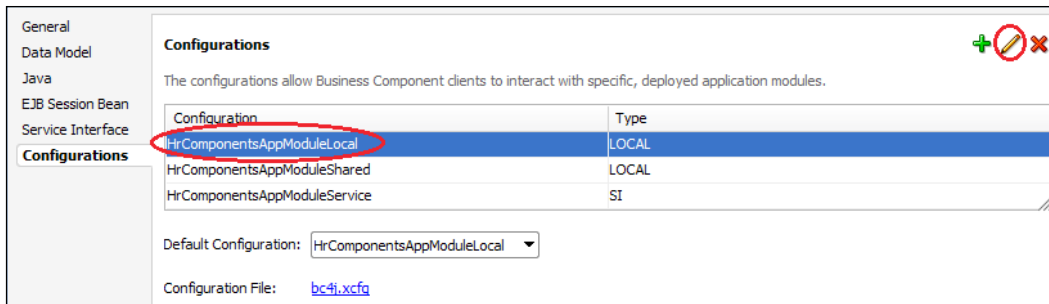
```
// return custom transaction framework
// extension implementation
return new ExtDBTransactionImpl2();
```

3. Load the `ExtDBTransactionImpl2.java` file in the Java editor, add an `ADFLogger` to it and override the `commit()` and `rollback()` `DBTransactionImpl2` methods. The code should look similar to the following:

```
// create an ADFLogger
private static final ADFLogger LOGGER =
    ADFLogger.createADFLogger(ExtDBTransactionImpl2.class);
public void commit() {
    // log a trace
    LOGGER.info("Commit was called on the transaction");
    super.commit();
}
public void rollback() {
    // log a trace
    LOGGER.info("Rollback was called on the transaction");
    super.rollback();
}
```

4. Rebuild and redeploy the `SharedComponents` workspace into an ADF Library JAR.
5. Open the `HRComponents` workspace and open the `HrComponentsAppModule` application module definition by double-clicking on it in the **Application Navigator**.

6. Go to the **Configurations** section.
7. Select the **HrComponentsAppModuleLocal** configuration and click the **Edit selected configuration object** button (the pen icon).



8. In the **Edit Configuration** dialog, click on the **Properties** tab and locate the **TransactionFactory** property. For the property value, enter the custom transaction framework extension class `com.packt.jdeveloper.cookbook.shared.bc.extensions.ExtDatabaseTransactionFactory`.

How it works...

In steps 1 and 2, we have overridden the `DatabaseTransactionFactory` `create()` method for the custom transaction factory framework class `ExtDatabaseTransactionFactory` that we created in the recipe *Setting up BC base classes* recipe in *Chapter 1, Pre-requisites to Success: ADF Project Setup and Foundations*. Now it will return our custom transaction implementation class `ExtDBTransactionImpl2`. This informs the ADF Business Components framework that a custom `oracle.jbo.server.DBTransaction` implementation will be used. Then, in step 3, we provide custom implementations for our `ExtDBTransactionImpl2` transaction commit and rollback operations. In this case, we have provided a global transaction logging facility for all commit and roll back operations throughout the ADF application for application modules utilizing our custom `DBTransaction` implementation. We then rebuild and redeploy the shared components workspace (step 4).

In steps 5 through 8, we have explicitly indicated in the `HrComponentsAppModule` local configuration, the one used to configure session-specific application modules, that a custom transaction factory will be used. We did this by setting the `TransactionFactory` configuration property to our custom transaction factory implementation class `com.packt.jdeveloper.cookbook.shared.bc.extensions.ExtDatabaseTransactionFactory`.

There's more...

In order to change application module configuration parameters for all application modules throughout your ADF application, adopt the practice of using Java system-defined properties via the `-D` switch at JVM startup. In this case, ensure that no specific configuration parameters are defined for individual application modules, unless needed, as they would override the values specified globally with the `-D` Java switch. You can determine the specific parameter names that you must specify with the `D` switch, from the **Property** field in the **Edit Configuration** dialog. For example, for this recipe you will specify `DTransactionFactory="com.packt.jdeveloper.cookbook.shared.bc.extensions.ExtDatabaseTransactionFactory"` at JVM startup, to indicate that the custom transaction factory will be used. For WebLogic, these startup parameters can be specified via the `JAVA_OPTIONS` environment variable or in any of the WebLogic startup scripts (`setDomainEnv.*`, `startWebLogic.*`, `startManagedWebLogic.*`). These scripts can be found in the `bin` directory under the WebLogic domain directory. Furthermore, the WebLogic server Java startup parameters can be specified using the administrator console in the server **Configuration | Server Start** tab.

See also

- ▶ *Setting up BC base classes, Chapter 1, Pre-requisites to Success: ADF Project Setup and Foundations*
- ▶ *Overriding `remove()` to delete associated children entities, Chapter 2, Dealing with Basics: Entity Objects*

6

Go with the Flow: Task Flows

In this chapter, we will cover:

- ▶ Using an application module function to initialize a page
- ▶ Using a task flow initializer to initialize a task flow
- ▶ Calling a task flow as a URL programmatically
- ▶ Retrieving the task flow definition programmatically using MetadataService
- ▶ Creating a train

Introduction

Task flows are used for designing the ADF Fusion web application's control flow. They were introduced with the advent of the JDeveloper 11g R1 release as an alternative to standard JSF navigation flows. As such, they allow for the decomposition of monolithic application navigation flows (as in the case of JSF navigation flows) into modular, transaction, and memory scope aware controller flow components. The ADF Fusion web application is now composed of numerous task flows, called bounded task flows, usually residing in various ADF Library JARs, calling each other in order to construct the application's overall navigation flow.

In the traditional JSF navigation flow, navigation occurs between pages. Task flows introduce navigation between activities. A task flow activity is not necessarily a visual page component (view activity) as in the case of JSF navigation flows. It can be a call to Java code (method call activity), the invocation of another task flow (task flow call activity), a control flow decision (router activity), or something else. This approach provides a high degree of flexibility, modularity, and reusability when designing the application's control flow.

Using an application module function to initialize a page

A common use case when developing an ADF Fusion web application is to perform some sort of initialization before a particular page of the application is shown. Such an initialization could be: the creation of a new view object row, which will in effect place the view object in insert mode; the execution of a view object query, which could populate a table on the page; the execution of a database stored procedure; or something similar. This can easily be accomplished by utilizing a method call activity.

In this recipe, we will demonstrate the usage of the method call task flow activity by implementing the familiar use case of placing a web page in insert mode. Before the page is presented, a custom application module method (implemented in another workspace) will be called to place the view object in insert mode.

Getting ready

You will need a skeleton **Fusion Web Application (ADF)** workspace created before you proceed with this recipe. For this, we have used the `MainApplication` workspace that was developed in the *Breaking up the application in multiple workspaces* recipe in *Chapter 1, Prerequisites to Success: ADF Project Setup and Foundations*.

The recipe also uses the `HrComponents` workspace, which was created in the *Overriding remove() to delete associated children entities* recipe in *Chapter 2, Dealing with Basics: Entity Objects*.

Both the `HrComponents` and `MainApplication` workspaces require database connections to the HR schema.

How to do it...

1. Open the `HrComponents` workspace in JDeveloper.
2. Load the `HrComponentsAppModuleImpl` custom application module implementation class into the Java editor and add the following `prepare()` method to it:

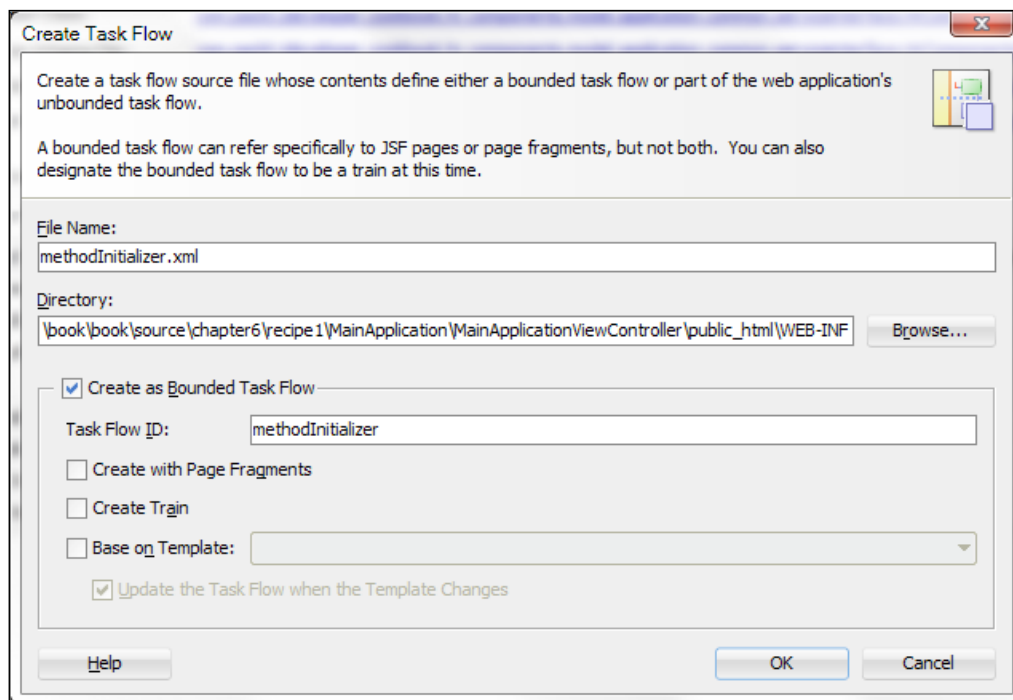
```
public void prepare() {
    // get the Employees view object instance
    EmployeesImpl employees = this.getEmployees();
    // remove all rows from rowset
    employees.executeEmptyRowSet();
    // create a new employee row
```

```

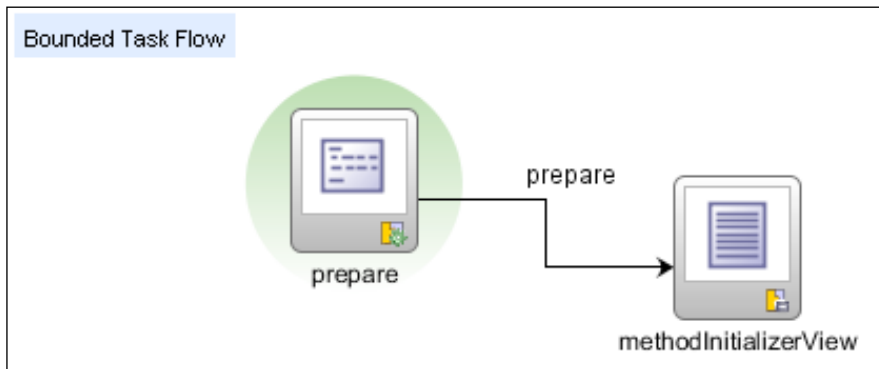
Row employee = employees.createRow();
// add the new employee to the rowset
employees.insertRow(employee);
}

```

- Open the `HrComponentsAppModule` application module definition and go to the **Java** section. Click on the **Edit application module client interface** button (the pen icon) and shuttle the `prepare()` method from the **Available** methods list to the **Selected** list.
- Rebuild and redeploy the `HrComponents` workspace into an ADF Library JAR.
- Now, open the `MainApplication` workspace and using the **Resource Palette** create a new **File System** connection to the `ReusableJARs` directory where the `HrComponents.jar` ADF Library JAR is placed. Select the `ViewController` project in the **Application Navigator** and then right-click on the `HrComponents.jar` ADF Library JAR in the **Resource Palette**. From the context menu, select **Add to Project...**
- Right-click on the `ViewController` project in the **Application Navigator** and select **New...** Select **ADF Task Flow** from the **Web Tier | JSF/Facelets** category.
- In the **Create Task Flow** dialog, enter `methodInitializer.xml` for the task flow name and ensure that you have selected the **Create as Bounded Task Flow** checkbox. Also, make sure that the **Create with Page Fragments** checkbox is not selected. Then click **OK**.



8. The `methodInitializer` task flow should open automatically in **Diagram** mode. If not, double-click on it in the **Application Navigator** to open it. Click anywhere in the task flow and in the **Property Inspector** change the **URL Invoke** property to **url-invoke-allowed**.
9. Expand the **Data Controls** section in the **Application Navigator**; locate and expand the `HrComponentsAppModuleDataControl` data control. Find the `prepare()` method and drag-and-drop it onto the `methodInitializer` task flow. JDeveloper will create a method call activity called `prepare`.
10. Drag-and-drop a **View** activity from the **Component Palette** onto the `methodInitializer` task flow.
11. Using the **Component Palette**, create a **Control Flow Case** from the `prepare` method call activity to the view activity.
12. Ensure that the `prepare` method call activity is marked as the default task flow activity by clicking on the **Mark Default Activity** button in the toolbar. The task flow should look similar to the following screenshot:



13. Double-click on the view activity to bring up the **Create JSF Page** dialog. In it, select **JSP XML** for the **Document Type**. For the **Page Layout**, you may select any of the **Quick Start Layout** options. Click **OK**. The page should open automatically in **Design** mode. If not, double-click on it in the **Application Navigator** to open it.

14. Expand the **Data Controls** section in the **Application Navigator** and locate the `Employees` view object under the `HrComponentsAppModuleDataControl`. Drag-and-drop the `Employees` view object onto the page.
15. From the **Create** context menu, select **Form | ADF Form...**. This will present the **Edit Form Fields** dialog. Click **OK** to accept the defaults and proceed with the creation of the ADF form.

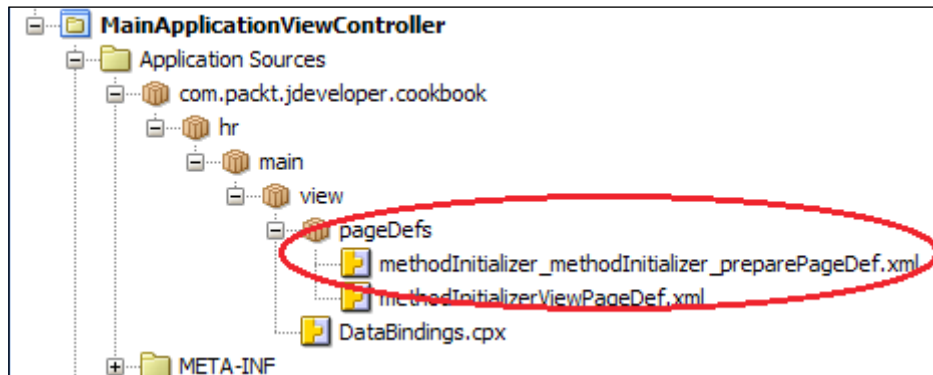
How it works...

In steps 1 through 4, we have added a method called `prepare()` to the `HrComponentsAppModule` application module residing in the `HRComponents` workspace. In this method, we retrieved the `Employees` view object instance by calling the `getEmployees()` method, and called `executeEmptyRowSet()` on the view object to empty its rowset. We then created an employee row by calling `createRow()` on the `Employees` view object, and added the new row to the `Employees` view object rowset by calling `insertRow()` and passing the newly created employee row as an argument to it. This will, in effect, place the `Employees` view object in insert mode. We exposed the `prepare()` method to the application module client interface (in step 3), so that we will be able to call this method via the bindings layer using a task flow method call activity. Then (in step 4), we rebuild and redeployed the `HRComponents` workspace to an ADF Library JAR. This will allow us to import the ADF components implemented in the ADF Library JAR to other projects throughout the ADF application.

In order to be able to reuse the components defined and implemented in the `HRComponents` ADF Library JAR, we created a file system connection using the **Resource Palette** in JDeveloper and added the library to our main project (in step 5).

In steps 6 through 8, we created a bounded task flow called `methodInitializer` and ensured (in step 8) that its **URL Invoke** property was set to `url-invoke-allowed`. We needed to do this because the method call activity that is added in step 9 to call the `prepare()` method in the `HrComponentsAppModule` application module is indicated as the default task flow activity (in step 12). In this case, leaving the default setting of `calculated` for the **URL Invoke** property will produce an *HTTP 403 Forbidden* error. This is a security precaution to disallow URL-invoking a task flow that does not have a view activity as its default activity. In our case, as we have indicated a method call activity as the default activity, we need to ensure that the **URL Invoke** property is set to `url-invoke-allowed`.

In step 9, we dragged-and-dropped the `prepare()` method, under the `HrComponentsAppModuleDataControl` data control (in the **Data Controls** section of the **Application Navigator**), onto the task flow. This creates the method call activity and the necessary bindings to bind the method call activity to the `prepare()` method in the `HrComponentsAppModule` application module. By default, the page definition is placed in the `pageDefs` package under the default package defined for the ViewController project (`com.packt.jdeveloper.cookbook.hr.main.view` in this case). Note that the `HrComponentsAppModuleDataControl` data control becomes available once the `HrComponents ADF Library JAR` is added to the project.



In steps 10 and 11, we placed a view activity onto the task flow and added a control flow case (called `prepare` by default) to allow the transition from the `prepare()` method call activity to the view activity.

The definition of the task flow is completed by ensuring that the `prepare()` method call activity is marked as the default task flow activity (step 12). This indicates that it will be the first activity to be executed in the task flow.

Finally, in steps 13 through 15, we create a JSF page for the task flow view activity and add an ADF form to it for the `Employees` view object. We did this by dragging-and-dropping the `Employees` view object from the `HrComponentsAppModuleDataControl` data control onto the JSPX page.

To test the recipe, right-click on the `methodInitializer` task flow in the **Application Navigator** and select **Run** or **Debug** from the context menu. This will build, deploy, and run the workspace into the integrated WebLogic application server. As you can see, the `prepare()` method call activity is called prior to transitioning to the view activity. The effect of calling the `prepare()` method is to place the `Employees` view object in insert mode.

The screenshot shows a web browser window with the address bar displaying `127.0.0.1:7101/MainApplication-MainApplication\`. The page title is `methodInitializerView.jspx`. The form contains the following fields and controls:

- `* EmployeeId`: Text input field with an asterisk indicating it is required.
- `FirstName`: Text input field.
- `* LastName`: Text input field with an asterisk indicating it is required.
- `* Email`: Text input field with an asterisk indicating it is required.
- `PhoneNumber`: Text input field.
- `* HireDate`: Text input field with the value `28/07/2011` and an asterisk indicating it is required.
- `* JobId`: Text input field with an asterisk indicating it is required.
- `Salary`: Text input field.
- `CommissionPct`: Text input field.
- `ManagerId`: Text input field.
- `DepartmentId`: Text input field.
- `LovAttrib`: Dropdown menu.
- `FavoriteColor`: Dropdown menu.

At the bottom of the form, there are five buttons: `First`, `Previous`, `Next`, `Last`, and `Submit`.

There's more...

Using a task flow method call activity is one of the possible ways to perform an initialization before displaying a page. Another approach, explained in more detail in the *Using a task flow initializer for task flow initialization* recipe in this chapter, is to use a task flow initializer. The difference between the two is that the task flow initializer is called once during the instantiation of the task flow, while multiple method call activities may be placed anywhere in the task flow. Also, consider the definition of an `invokeAction` in the page definition to perform page-specific initialization. As best practice, consider using either a method call task flow activity or a task flow initializer as they are highly abstracted and loosely coupled. Using an `invokeAction` on the other hand would be more appropriate if you want the initialization method to be executed for multiple phases of the page's lifecycle.

Furthermore, note that the `prepare()` method we use for initializing the page, does not accept any parameters and it returns nothing. If your initialization method requires parameters to be specified, they can be specified either in the **Parameters** section of the **Edit Action Binding** dialog (for a data control bound method) or otherwise, using the **Parameters** section in the method call **Property Inspector**. In either case, the parameter values are usually communicated via the `pageFlowScope`. Finally, based on the return type of your initialization method, you could set the value of the `toString()` outcome in the **Outcome** section of the **Property Inspector** and allow further processing of the return value, using a router activity for instance. When returning `void`, the outcome must be fixed and `toString()` cannot be used (must be set to `false`).

See also

- ▶ *Breaking up the application in multiple workspaces, Chapter 1, Pre-requisites to Success: ADF Project Setup and Foundations*
- ▶ *Overriding `remove()` to delete associated children entities, Chapter 2, Dealing with Basics: Entity Objects*
- ▶ *Using a task flow initializer to initialize a task flow, in this chapter*

Using a task flow initializer to initialize a task flow

In the *Using an application module function to initialize a page* recipe in this chapter, we demonstrated how to use a method residing in the application module to perform page initialization, by bounding the method as a method call activity in the task flow. This recipe shows a different way to accomplish the same task by using a task flow initializer method instead. Unlike the method call activity, which once bound to the task flow may be called multiple times in the task flow, the initializer method is called only once during the task flow initialization.

Getting ready

You will need a skeleton **Fusion Web Application (ADF)** workspace created before you proceed with this recipe. For this, we have used the `MainApplication` workspace that was developed in the *Breaking up the application in multiple workspaces* recipe in *Chapter 1, Pre-requisites to Success: ADF Project Setup and Foundations*.

The recipe also uses the `HRComponents` workspace, which was created in the *Overriding `remove()` to delete associated children entities* recipe in *Chapter 2, Dealing with Basics: Entity Objects*.

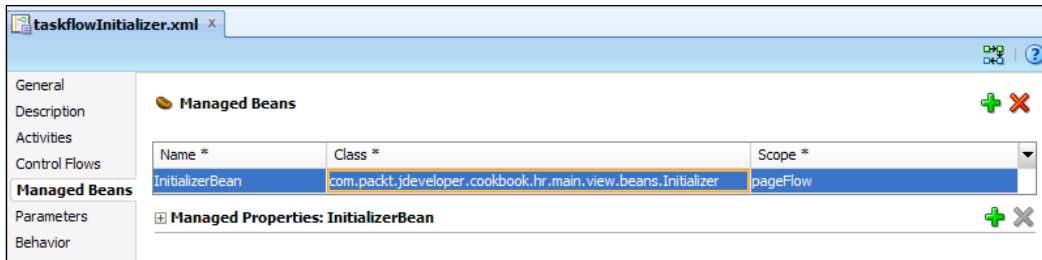
Both the `HRComponents` and `MainApplication` workspaces require database connections to the HR schema.

How to do it...

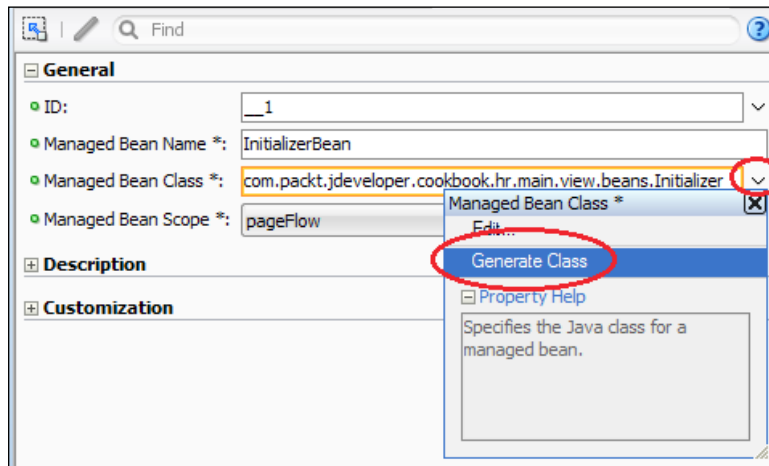
1. Open the `HRComponents` workspace in JDeveloper.
2. Load the `HrComponentsAppModuleImpl` custom application module implementation class into the Java editor and add the following method to it:


```
public void prepare() {
    // get the Employees view object instance
    EmployeesImpl employees = this.getEmployees();
    // remove all rows from rowset
    employees.executeEmptyRowSet();
    // create a new employee row
    Row employee = employees.createRow();
    // add the new employee to the rowset
    employees.insertRow(employee);
}
```
3. Open the `HrComponentsAppModule` application module definition and go to the **Java** section. Click on the **Edit application module client interface** button (the pen icon) and shuttle the `prepare()` method from the **Available** methods list to the **Selected** list.
4. Rebuild and redeploy the `HRComponents` workspace into an ADF Library JAR.
5. Now, open the `MainApplication` workspace and using the **Resource Palette** create a new **File System** connection to `ReUsableJARs` directory where the `HRComponents.jar` ADF Library JAR is placed. Select the `ViewController` project in the **Application Navigator** and then right-click on the `HRComponents.jar` ADF Library JAR in the **Resource Palette**. From the context menu, select **Add to Project...**
6. Right-click on the `ViewController` project in the **Application Navigator** and select **New...** Select **ADF Task Flow** from the **Web Tier | JSF/Facelets** category.
7. In the **Create Task Flow** dialog, enter `taskflowInitializer.xml` for the task flow **File Name** and ensure that you have selected the **Create as Bounded Task Flow** checkbox. Also make sure that the **Create with Page Fragments** checkbox is not selected. Then click **OK**.

- The `taskflowInitializer` task flow should open automatically in **Diagram** mode. If not, double-click on it in the **Application Navigator** to open it. Click anywhere in the task flow, then in the **Property Inspector** change the **URL Invoke** property to **url-`invoke-allowed`**.
- Go to the task flow **Overview | Managed Beans** section and add a managed bean called `InitializerBean`. Enter `com.packt.jdeveloper.cookbook.hr.main.view.beans.Initializer` for the managed bean **Class** and select **pageFlow** for the bean **Scope**.



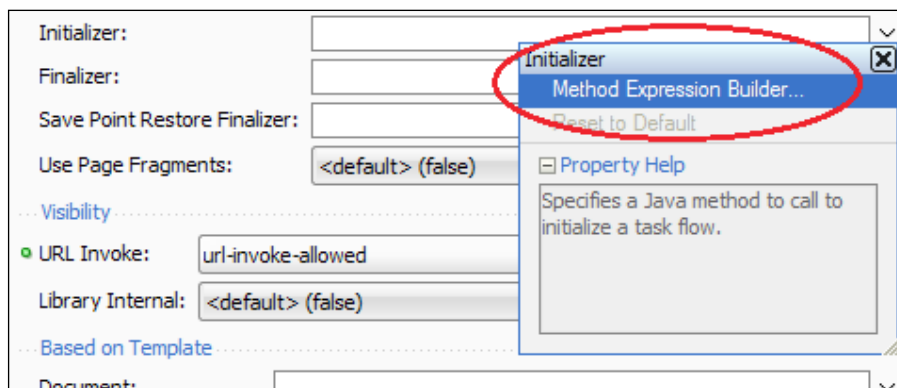
- While at the **Managed Beans** section, select **Generate Class** from the **Property Menu** next to the **Managed Bean Class** in the **Property Inspector**.



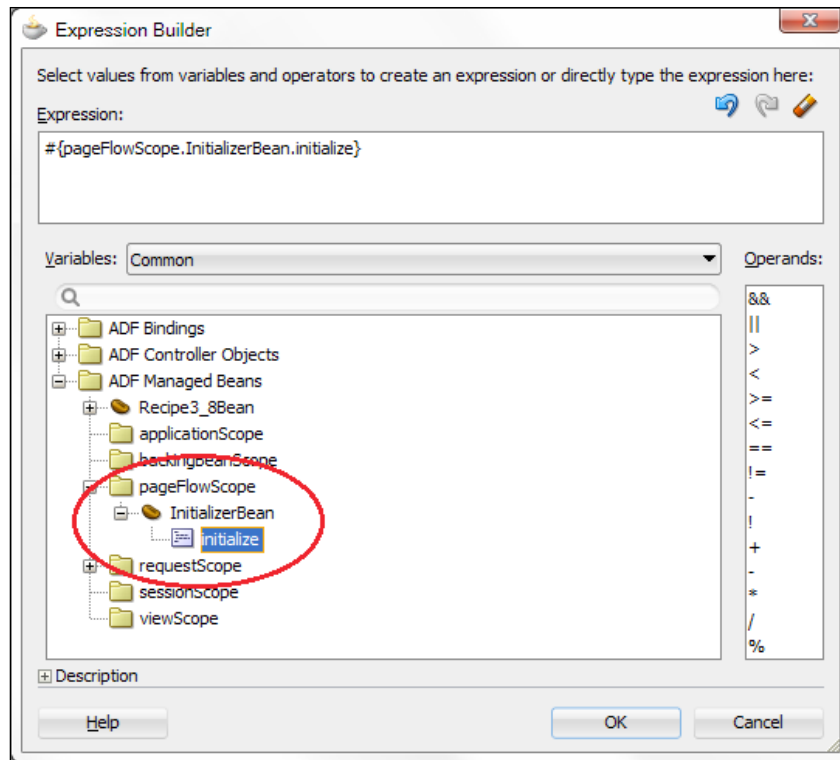
11. Locate the `Initializer.java` bean in the **Application Navigator** and open it in the Java editor. Add the following `initialize` method to it:

```
public void initialize() {  
    // get the application module  
    HrComponentsAppModule hrComponentsAppModule =  
        (HrComponentsAppModule)ADFUtils  
        .getApplicationModuleForDataControl(  
            "HrComponentsAppModuleDataControl");  
    if (hrComponentsAppModule != null) {  
        // call the initializer method  
        hrComponentsAppModule.prepare();  
    }  
}
```

12. Return to the task flow, diagram and add a task flow initializer by clicking on the **Property Menu** next to the **Initializer** property in the **Property Inspector** and selecting **Method Expression Builder...**



In the **Expression Builder** dialog that opens, locate and select the **initialize** method of the **InitializerBean** under the **ADF Managed Beans** node. The click **OK** to dismiss the dialog. The initializer expression `# {pageFlowScope.InitializerBean.initialize}` should be reflected in the **Initializer** property of the task flow in the **Property Inspector**.



13. Drag-and-drop a **View** activity from the **Component Palette** onto the `taskflowInitializer` task flow.
14. Double-click on the view activity to bring up the **Create JSF Page** dialog. In it, select **JSP XML** for the **Document Type**. For the **Page Layout**, you may select any of the **Quick Start Layout** options. Click **OK**. The page should open automatically in **Design** mode. If not, double-click on it in the **Application Navigator** to open it.
15. Expand the **Data Controls** section in the **Application Navigator** and locate the Employees view object under the `HrComponentsAppModuleDataControl`. Drag-and-drop the Employees view object onto the page.
16. From the **Create** context menu, select **Form | ADF Form....** This will present the **Edit Form Fields** dialog. Click **OK** to accept the defaults and proceed with the creation of the ADF form.

How it works...

Steps 1 through 8 have been thoroughly explained in the *Using an application module function to initialize a page* recipe in this chapter, so we won't get into the specific details here.

In steps 9 and 10, we defined a managed bean called `InitializerBean` and generated a Java class for it. We used `pageFlow` for the bean's memory scope. This ensures that the `InitializerBean` bean persists throughout the task flow's execution.

In step 11, we added an `initialize()` method to the `InitializerBean` bean. This is the method indicated as the task flow initializer in steps 12 and 13. Inside the `initialize()` method, we get hold of the `HrComponentsAppModule` by utilizing the `ADFUtils.getApplicationModuleForDataControl()` helper method. We introduced the `ADFUtils` helper class back in *Chapter 1, Pre-requisites to Success: ADF Project Setup and Foundations* in the *Using ADFUtils/JSFUtils* recipe. We have packaged the `ADFUtils` helper class inside the `SharedComponents` ADF Library JAR (`SharedComponents.jar`), which is imported into the project in step 5. The `getApplicationModuleForDataControl()` method returns an `oracle.jbo.ApplicationModule` interface, which we then cast to our specific `HrComponentsAppModule` custom application module interface. Through the `HrComponentsAppModule` interface, we call the `prepare()` method to do the necessary initializations. We explained the logic in `prepare()` in the *Using an application module function to initialize a page* recipe in this chapter.

In steps 12 and 13, we declaratively setup the task flow initializer property using the Expression Language expression `#{pageFlowScope_INITIALIZERBean.initialize}`. This expression indicates that the `initialize()` method of the `InitializerBean` is called during the instantiation of the task flow.

Finally, in steps 14 through 17, we defined a view activity and the corresponding JSF page. Again, we explained these steps in more detail in the *Using an application module function to initialize a page* recipe in this chapter.

To test the recipe, right-click on the `taskFlowInitializer` task flow in the **Application Navigator** and select **Run** from the context menu. This will build, deploy and run the workspace into the integrated WebLogic application server. The page displayed in the browser will be presented in insert mode, as the task flow initializer method calls the application module `prepare()` method to set the `Employees` view object in insert mode.

There's more...

Both this technique and the one presented in the *Using an application module function to initialize a page* recipe in this chapter may be used to run task flow initialization code. However, note one difference pertaining to their handling of the Web browser's back button. While the task flow initializer approach calls the initializer method upon reentry via the browser's back button, no task flow initialization code is called when reentering the task flow via the browser's back button in the method call activity approach. However, this behaviour seems to be inconsistent among browsers, depending on how they handle page caching. For more information about this, refer to section *About Creating Complex Task Flows* in the *Fusion Developer's Guide for Oracle Application Framework* which can be found at http://docs.oracle.com/cd/E24382_01/web.1112/e16182/toc.htm.

See also

- ▶ *Breaking up the application in multiple workspaces, Chapter 1, Pre-requisites to Success: ADF Project Setup and Foundations*
- ▶ *Overriding `remove()` to delete associated children entities, Chapter 2, Dealing with Basics: Entity Objects*
- ▶ *Using an application module function to initialize a page*, in this chapter

Calling a task flow as a URL programmatically

A task flow that is indicated as URL invocable (by setting its `visibility` attribute `url-invoke-allowed` to `true`) may be accessed directly by constructing and invoking its URL. This allows you to dynamically invoke task flows from within your Java code depending on some condition that is satisfied at runtime. Programmatically, this can be done using the `oracle.adf.controller.ControllerContext.getTaskFlowURL()` method and specifying the task flow identifier and parameters.

For this recipe, to demonstrate calling a task flow via its URL, we will create a task flow that is URL invocable and call it from a JSF page programmatically. The task flow accepts a parameter and based on the parameter's value, determines whether to call any of the `methodInitializer` or `taskflowInitializer` task flows. These task flows were developed in the *Using an application module function to initialize a page* and *Using a task flow initializer to initialize a task flow* recipes respectively in this chapter.

Getting ready

You need to have access to the `methodInitializer` and `taskflowInitializer` task flows that were developed in the *Using an application module function to initialize a page* and *Using a task flow initializer to initialize a task flow* recipes in this chapter. Also, note the additional prerequisites stated for those recipes, that is, the usage of the `HRComponents` and `MainApplication` workspaces and the database connection to the HR schema.

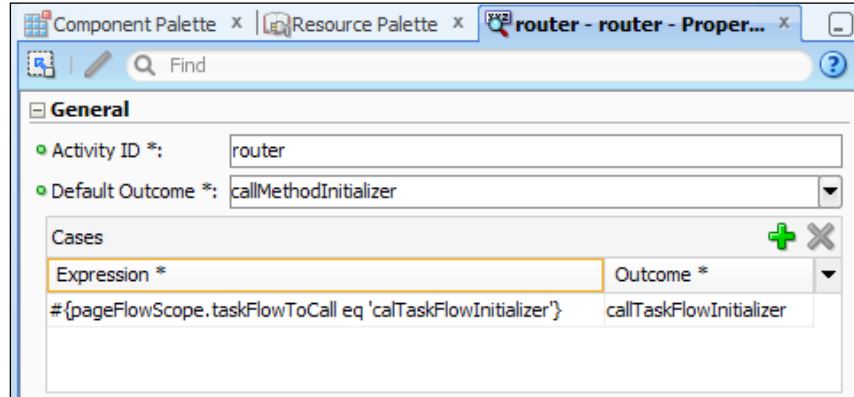
How to do it...

1. Start by creating a new task flow called `programmaticallyInvokeTaskFlow`. Ensure that you create it as a bounded task flow, and that it is not created with page fragments.
2. In the **Visibility** section in the task flow **Property Inspector**, make sure that the **URL Invoke** attribute is set to `url-invoke-allowed`.
3. While in the task flow **Property Inspector**, in the **Parameters** section add a parameter called `taskFlowToCall` of type `java.lang.String`. For the parameter **Value** enter `{pageFlowScope.taskFlowToCall}`.

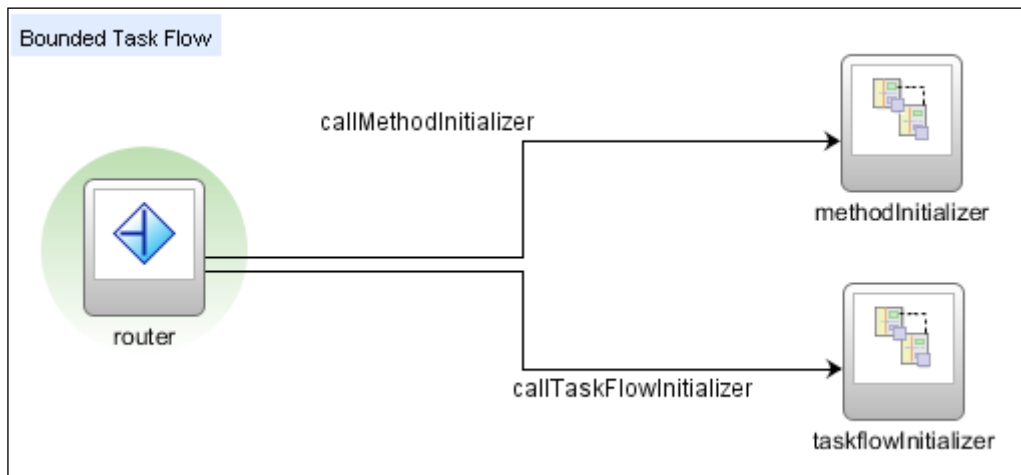
Name *	Class	Value	Required
taskFlowToCall	java.lang.String	{pageFlowScope.taskFlowToCall}	<input type="checkbox"/>

4. From the **Component Palette**, drop a **Router** activity on the task flow.
5. Locate the `methodInitializer` and `taskflowInitializer` task flows in the **Application Navigator** and drop them on the task flow.
6. Using the **Component Palette**, create control flow cases from the router activity to `methodInitializer` and `taskflowInitializer` task flow calls. Call these control flow cases `callMethodInitializer` and `callTaskFlowInitializer` respectively.

- Next select the router activity, and in the **Property Inspector** set its **Default Outcome** property to `callMethodInitializer`. Also, add the following expression `#{pageFlowScope.taskFlowToCall eq 'callTaskFlowInitializer'}` in the **Cases** section and `callTaskFlowInitializer` as the expression's **Outcome** value. The router's properties in the Property Inspector should look similar to the following screenshot:



- The complete `programmaticallyInvokeTaskFlow` task flow should look similar to the following screenshot:



- Now, locate the `adfc-config.xml` unbounded task flow in the **Application Navigator** and double-click on it to open it. Go to the **Overview | Managed Beans** section and add a `TaskFlowURLCallerBean` managed bean. Specify `com.packt.jdeveloper.cookbook.hr.main.view.beans.TaskFlowURLCaller` for the bean **Class** and leave the default **request** for the bean's **Scope**.

10. Create the managed bean by selecting **Generate Class** from the **Property Menu** in the **Property Inspector**, next to the **Managed Bean Class** attribute.
11. Locate the `TaskFlowURLCallerBean` bean in the **Application Navigator** and double-click on it to open it in the Java editor. Add the following methods to it:

```
public String getProgrammaticallyInvokeTaskFlow() {
    // setup task flow parameters
    Map<String, Object> parameters =
        new java.util.HashMap<String, Object>();
    parameters.put("taskFlowToCall", "callTaskFlowInitializer");
    // construct and return the task flow's URL
    return getTaskFlowURL("/WEB-INF/programmaticallyInvokeTaskFlow.xml
        #programmaticallyInvokeTaskFlow", parameters);
}

private String getTaskFlowURL(String taskFlowSpecs, Map<String,
Object> parameters) {
    // create a TaskFlowId from the task flow specification
    TaskFlowId tfid = TaskFlowId.parse(taskFlowSpecs);
    // construct the task flow URL
    String taskFlowURL =
        ControllerContext.getInstance().getTaskFlowURL(
            false, tfid, parameters);
    // remove the application context path from the URL
    FacesContext fc = FacesContext.getCurrentInstance();
    String taskFlowContextPath =
        fc.getExternalContext().getRequestContextPath();
    return taskFlowURL.replaceFirst(taskFlowContextPath, "");
}
```

12. Finally, create a JSPX page called `taskFlowURLCaller.jspx` and drop a **Link (Go)** component on it from the **Component Palette**. Specify the link's text, destination, and `targetFrame` properties as follows:

```
<af:goLink text="Call programmaticallyInvokeTaskFlow as a URL"
    id="gl1"
    destination="#{TaskFlowURLCallerBean.
        programmaticallyInvokeTaskFlow}"
    targetFrame="_blank"/>
```

How it works...

In steps 1 through 3, we created a task flow called `programmaticallyInvokeTaskFlow` and set its visibility to `url-invoke-allowed`. This allows us to call the task flow via a URL. If we don't do this, a security exception will be thrown when trying to access the task flow via a URL. This was discussed in greater detail in recipe *Using an application module function to initialize a page* in this chapter. We also added (in step 3), a single task flow parameter called `taskFlowToCall` to indicate which task flow to call once our `programmaticallyInvokeTaskFlow` is executed. We stored the value of this parameter to a pageFlow scope variable called `taskFlowToCall`. This parameter is accessible via the EL expression `#{pageFlowScope.taskFlowToCall}`. We will see in step 7 how this pageFlow scope variable is accessed to determine the subsequent task flow to call.

In steps 4 through 8, we completed the task flow definition by adding a router activity and two task flow call activities, one for each of the `callMethodInitializer` and `callTaskFlowInitializer` task flows. Note, in step 5, how we just dropped the `callMethodInitializer` and `callTaskFlowInitializer` task flows from the **Application Navigator**, to create the task flow calls. Also, observe in step 6, how we have created the control flow cases to connect the router activity with each of the task flow call activities. Finally, note how in step 7, we configured the router activity outcomes based on the value of the input task flow parameter `taskFlowToCall`. Specifically, we checked the parameter's value using the EL expression `#{pageFlowScope.taskFlowToCall eq 'callTaskFlowInitializer'}`. In this case, the router's outcome was set to `callTaskFlowInitializer`, which calls the `taskflowInitializer` task flow. In any other case, we configured the default router outcome to be `callMethodInitializer`, which calls the `methodInitializer` task flow.

In steps 9 through 11, we configured a globally accessible managed bean called `TaskFlowURLCallerBean`, by adding it to the application's unbounded task flow `adfc-config.xml`. We generated the bean class in step 10 and 11, where we added the necessary code to be able to call our `programmaticallyInvokeTaskFlow` task flow programmatically. The specific details about this code follow.

We introduced two methods in the `TaskFlowURLCallerBean`. One called `getProgrammaticallyInvokeTaskFlow()`, which will be called from a page component to return the task flow's URL (see step 12) and another one called `getTaskFlowURL()`, a helper method to do the actual work of determining and returning the task flow's URL. We call `getTaskFlowURL()` indicating the task flow specification and its parameters.

Observe in `getProgrammaticallyInvokeTaskFlow()`, how we specify the parameter value and the task flow specifications. In `getTaskFlowURL()`, we obtain an `oracle.adf.controller.TaskFlowId` from the task flow identifier, and then call the `oracle.adf.controller.ControllerContext.getTaskFlowURL()` method to retrieve the task flow URL. Once the URL is returned, we strip the application's context path from it before returning it. This is something that we need to do before calling the task flow via a URL because the application context path should not be part of the task flow URL when invoking the task flow. The final format of the task flow URL returned by `getTaskFlowURL()` looks something similar to `/faces/adf.task-flow?adf.tfDoc=/WEB-INF/programmaticallyInvokeTaskFlow.xml&adf.tfId=programmaticallyInvokeTaskFlow&taskFlowToCall=callTaskFlowInitializer`.

The final part of the implementation is done in step 12. In this step, we created a new JSF page, called `taskFlowURLCaller.jspx`, and added an `af:goLink` ADF Faces UI component to it. We use the go link to programmatically call our `programmaticallyInvokeTaskFlow` task flow, via the URL returned by the `getProgrammaticallyInvokeTaskFlow()` method defined in the `TaskFlowURLCallerBean`. We do this by setting the destination attribute of the `af:goLink` component to `#{TaskFlowURLCallerBean.programmaticallyInvokeTaskFlow}`. We also indicate `_blank` for the go link `targetFrame` attribute, so that the called task flow opens in a new browser frame.

To test the recipe, right-click on the `taskFlowURLCaller.jspx` page in the **Application Navigator** and select **Run** or **Debug** from the context menu.

There's more...

When calling a task flow programmatically via its URL, always use the ADF Controller API indicated in this recipe to obtain the task flow's URL. Do not hardcode the task flow's URL in your application or in database tables, as the specifications of the task flow URL in the ADF framework (the task flow URL format) may change in the future.

See also

- ▶ *Using an application module function to initialize a page*, in this chapter
- ▶ *Using a task flow initializer to initialize a task flow*, in this chapter

Retrieving the task flow definition programmatically using MetadataService

Task flow definition in JDeveloper is done through the declarative support provided by the IDE. This includes defining the task flow activities and their relevant control flow cases by dragging-and-dropping task flow components from the **Component Palette** to the **Diagram** tab and adjusting their properties through the **Property Inspector**, defining managed beans in the **Overview** tab, and so on. JDeveloper saves the task flow definition metadata in an XML document, which is accessible in JDeveloper anytime you click on the **Source** tab. The task flow definition metadata is available programmatically at runtime through the `oracle.adf.controller.metadata.MetadataService` object by calling `getTaskFlowDefinition()`. This API is public since the release of JDeveloper version 11.1.2.

In this recipe, we will show how to get the task flow definition metadata by implementing the following use case. For each task flow in our ADF application, this will provide a generic technique for logging the task flow input parameters upon task flow entry and the task flow return values upon task flow exit.

Getting ready

You will need to have access to the `SharedComponents` workspace that was developed in the *Breaking up the application in multiple workspaces* recipe in *Chapter 1, Pre-requisites to Success: ADF Project Setup and Foundations*. New functionality will be added to the `ViewController` project that is part of the `SharedComponents` workspace.

Moreover, this recipe enhances the `taskflowInitializer` task flow developed in the *Using a task flow initializer to initialize a task flow* recipe in this chapter. Note the additional prerequisites stated for those recipes, that is, the usage of the `HRComponents` and `MainApplication` workspaces and the database connection to the HR schema.

How to do it...

1. Open the `SharedComponents` workspace and create a new Java class called `TaskFlowBaseBean`. Add the following methods to it:

```
public void initialize() {
    // get task flow parameters
    Map<String, TaskFlowInputParameter> taskFlowParameters =
        getTaskFlowParameters();
    // log parameters
    logParameters(taskFlowParameters);
}
public void finalize() {
    // get task flow return values
```

```

        Map<String, NamedParameter> taskFlowReturnValues =
            getReturnValues();
        // log return values
        logParameters(taskFlowReturnValues);
    }
    protected TaskFlowId getTaskFlowId() {
        // get task flow context from the current view port
        TaskFlowContext taskFlowContext =
            ControllerContext.getInstance().getCurrentViewPort()
                .getTaskFlowContext();
        // return the task flow id
        return taskFlowContext.getTaskFlowId();
    }
    protected TaskFlowDefinition getTaskFlowDefinition() {
        // use MetadataService to return the task flow
        // definition based on the task flow id
        return MetadataService.getInstance()
            .getTaskFlowDefinition(getTaskFlowId());
    }
    protected Map<String, TaskFlowInputParameter>
        getTaskFlowParameters() {
        // get task flow definition
        TaskFlowDefinition taskFlowDefinition =
            getTaskFlowDefinition();
        // return the task flow input parameters
        return taskFlowDefinition.getInputParameters();
    }
    protected Map<String, NamedParameter> getReturnValues() {
        // get task flow definition
        TaskFlowDefinition taskFlowDefinition =
            getTaskFlowDefinition();
        // return the task flow return values
        return taskFlowDefinition.getReturnValues();
    }
    public void logParameters(Map taskFlowParameters) {
        // implement parameter logging here
    }
}

```

2. Rebuild and redeploy the SharedComponents ADF Library JAR.
3. Open the MainApplication workspace and add the SharedComponents ADF Library JAR—deployed in the previous step—to its ViewController project.
4. Load the `InitializerBean` managed bean implementation class `com.packt.jdeveloper.cookbook.hr.main.view.beans.Initializer` into the Java editor, and change it so that it extends the `TaskFlowBaseBean` class:

```
public class Initializer extends TaskFlowBaseBean
```

5. Also, update its `initialize()` method by adding a call to `super.initialize()` and add the following `finalize()` method:

```
public void finalize() {  
    // allow base class processing  
    super.finalize();  
}
```

6. Finally, add a finalizer to the `taskflowInitializer` task flow using the following EL expression:

```
#{pageFlowScope_INITIALIZERBean.finalize}
```

How it works...

In step 1, we create a class called `TaskFlowBaseBean` that we can use throughout our ADF application as the base class from which beans providing task flow initializer and finalizer methods can be derived (as we did in step 3 in this recipe). This class consists of initializer and finalizer methods that retrieve and log the task flow input parameters and return values respectively. These methods are implemented by `initialize()` and `finalize()` and they are publicly accessible, which means that they can be directly used from within JDeveloper when defining task flow initializers and/or finalizers. This is useful if you don't want to provide any specific implementations of the task flow initializer and/or finalizer method. The `initialize()` method calls the helper `getTaskFlowParameters()` to retrieve the input task flow parameters and then calls `logParameters()` to log these parameters. Similarly, `finalize()` calls `getReturnValues()` to retrieve the returned values and `logParameters()` to log them. The `getTaskFlowParameters()` and `getReturnValues()` helper methods rely on getting the task flow definition `oracle.adf.controller.metadata.model.TaskFlowDefinition` object and calling `getInputParameters()` and `getReturnValues()` on it, respectively. The task flow definition is returned by the helper `getTaskFlowDefinition()`, which retrieves it by calling the `oracle.adf.controller.metadata.MetadataService` method `getTaskFlowDefinition()`. This method accepts an `oracle.adf.controller.TaskFlowId`, indicating the task flow identifier for which we are inquiring the task flow definition. We retrieve the current task flow identifier by calling the helper `getTaskFlowId()`, which retrieves the current task flow from the task flow context obtained from the current view port, as shown in the following lines of code:

```
// get task flow context from the current view port  
TaskFlowContext taskFlowContext =  
    ControllerContext.getInstance()  
        .getCurrentViewPort().getTaskFlowContext();  
// return the task flow id  
return taskFlowContext.getTaskFlowId();
```

In step 2, we re-deployed the `SharedComponents` workspace as an ADF Library JAR. Then, in step 3, we added it to the `MainApplication ViewController` project. One way to do this is through the **Resource Palette**.

To demonstrate the usage of the `TaskFlowBaseBean` class, we have updated the `InitializerBean` managed bean class `Initializer` that was developed in an earlier recipe, so that `TaskFlowBaseBean` class is derived from it (in step 4). Then (in step 5), we updated the `Initializer` class `initialize()` method to call `TaskFlowBaseBean`'s `initialize()` to do the base class processing, that is, to log any input parameters.

In steps 5 and 6, to complete the recipe, we added a task flow finalizer, which simply calls the base class' `super.finalize()` to log the returned task flow parameters.

There's more...

The implementation of `logParameters()`, not included in the book's source, is left as an exercise. This method should basically iterate over the task flow parameters and for each one obtain its value expression by calling the `oracle.adf.controller.metadata.model.Parameter.getValueExpression()` method. The parameter's value expression can be evaluated by calling the `javax.faces.application.Application.evaluateExpressionGet()` method.

Also, note that task flow metadata is loaded from ADF Controller metadata resources using the following search rules. Firstly, resources named `META-INF/adfc-config.xml` in the classpath are loaded and then the existence of the web application configuration resource named `/WEB-INF/adfc-config.xml` is checked and loaded if it exists. Once these resources are loaded, they may reference other metadata objects that reside in other resources. These ADF Controller metadata resources are used to construct a model for the unbounded task flow. Metadata for bounded task flows is loaded on demand.

For a complete reference to all the methods available by the `MetaDataService` and `TaskFlowDefinition` classes, consult the *Oracle Fusion Middleware Documentation Library 11g Release 2 Java API Reference for Oracle ADF Controller*. It can be found at the URL http://download.oracle.com/docs/cd/E16162_01/apirefs.1112/e17480/toc.htm.

Furthermore, consult the article *Programmatically capturing task flow parameters* by Chris Muir where he describes the topic in greater detail. It can be found at the URL <http://one-size-doesnt-fit-all.blogspot.com/2010/10/jdev-programmatically-capturing-task.html>.

See also

- ▶ *Breaking up the application in multiple workspaces, Chapter 1, Pre-requisites to Success: ADF Project Setup and Foundations*
- ▶ *Using a task flow initializer to initialize a task flow, in this chapter*

Creating a train

Wizard-like user interfaces can be created in ADF using task flows created as trains and ADF Faces user interface components, such as the `af:train` (Train) and `af:trainButtonBar` (Train Button Bar) components. Using such an interface, you are presented with individual steps, called train stops, in a multi-step process, each step being a task flow activity or a combination of activities. Options exist that allow for the configuration of the train stops, controlling the sequential execution of the train stops, whether a train stop can be skipped, and others. Furthermore, a train stop can incorporate other task flow activities, such as method calls. Other task flows themselves can be added as train stops in the train (as task flow call activities).

In this recipe, we will go over the creation of a train consisting of view, method call, and task flow call activities.

Getting ready

You will need a skeleton **Fusion Web Application (ADF)** workspace created before you proceed with this recipe. For this, we have used the `MainApplication` workspace that was developed in the *Breaking up the application in multiple workspaces* recipe in *Chapter 1, Pre-requisites to Success: ADF Project Setup and Foundations*.

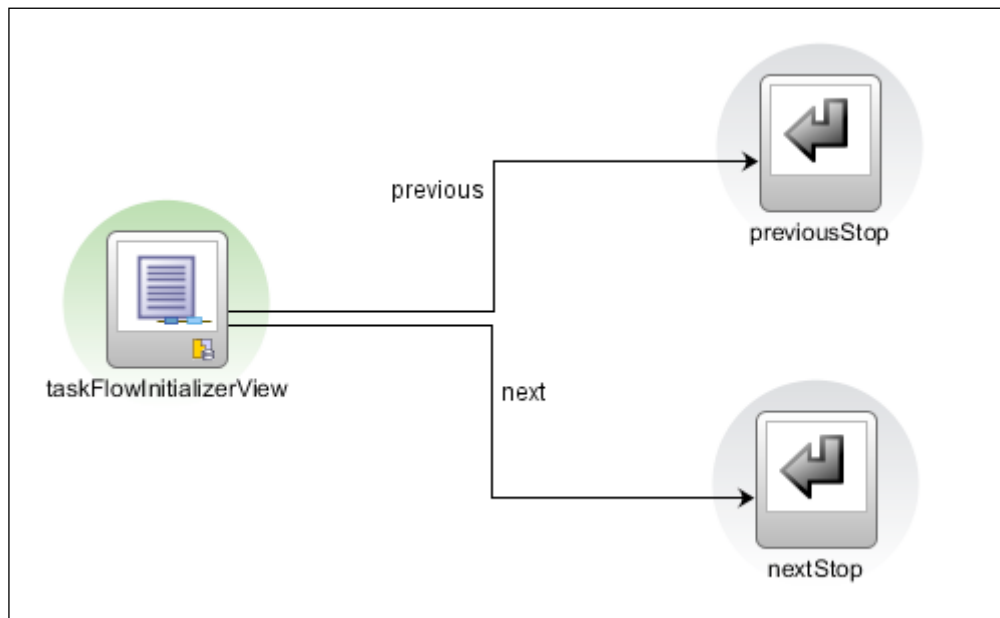
To demonstrate a method call activity as part of the train stop, the recipe uses the `HRComponents` workspace, which was created in the *Overriding `remove()` to delete associated children entities* recipe in *Chapter 2, Dealing with Basics: Entity Objects*. Moreover, to demonstrate a task flow call as a train stop, the recipe uses the `taskflowInitializer` task flow created in the *Using a task flow initializer to initialize task flow* recipe in this chapter.

Both the `HRComponents` and `MainApplication` workspaces require database connections to the HR schema.

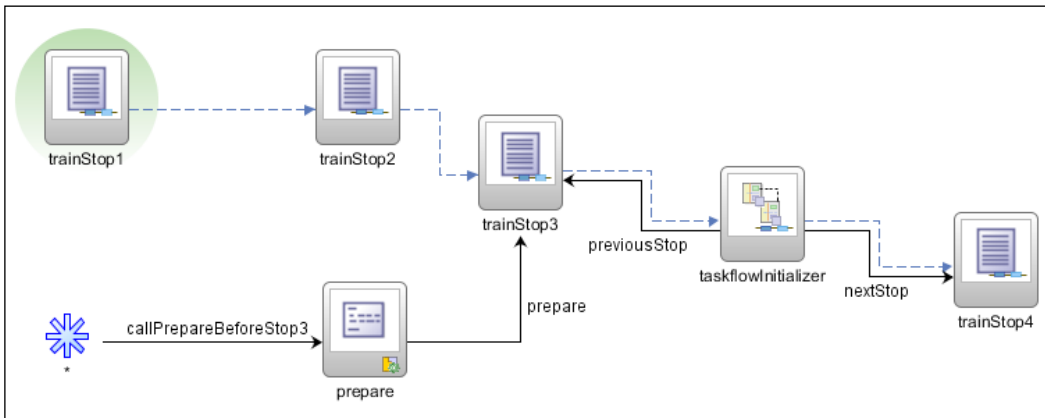
How to do it...

1. Create a bounded task flow called `trainTaskFlow`. Ensure that the **Create Train** checkbox in the **Create Task Flow** dialog is selected. We will not be using page fragments, so ensure that the **Create with Page Fragments** checkbox is not selected.

- From the **Component Palette** drop four view activities on the task flow. Call the view activities `trainStop1`, `trainStop2`, `trainStop3`, and `trainStop4`.
- Expand the **Data Controls** node in the **Application Navigator** and the `HrComponentsAppModuleDataControl`. Locate and drop the `prepare()` method on the task flow.
- Create a **Control Flow Case** from the `prepare()` method call to the `trainStop3` view activity.
- Drop a **Wildcard Control Flow Rule** from the **Component Palette** to the task flow and create a **Control Flow Case** called `callPrepareBeforeStop3` from the **Wildcard Control** to the `prepare()` method call.
- Select the `trainStop3` view activity and in the **Property Inspector** enter `callPrepareBeforeStop3` for its **Outcome** attribute.
- Locate the `taskflowInitializer` task flow in the **Application Navigator** and double-click on it to open it. From the **Component Palette**, drop two **Task Flow Return** components to it, called `previousStop` and `nextStop`.
- From the **Component Palette** add two **Control Flow Cases** and connect them from the `taskFlowInitializerView` view activity to the `previousStop` and `nextStop` task flow return activities. Call them `previous` and `next` respectively. The modified `taskflowInitializer` task flow should look similar to the following screenshot:



9. Return to the `trainTaskFlow` task flow. In the **Application Navigator**, locate the `taskflowInitializer` task flow and drop it in the `trainTaskFlow` task flow.
10. Right-click on the `trainStop4` view activity and select **Train | Move Backward** from the context menu.
11. Create two **Control Flow Cases** called `previousStop` and `nextStop` from the `taskflowInitializer` task flow call activity to the `trainStop3` and `trainStop4` view activities. This complete `taskflowInitializer` task flow should look similar to the following screenshot:



12. Now, double-click on each of the `trainStop1`, `trainStop2`, `trainStop3`, and `trainStop4` view activities in the `taskflowInitializer` task flow to create the JSF pages. In the **Create JSF** page dialog, select **JSP XML** for the **Document Type**.
13. For each of the pages created, select a **Train** component from the **ADF Faces Component Palette** and drop them on the pages. On the **Bind train** dialog that is displayed, accept the default binding and click **OK**.

