Oracle® Application Development Framework

Case Manual

10*g* Release 2 (10.1.2)

B19163-01

July 2005

Oracle Application Development Framework Case Manual, 10*g* Release 2 (10.1.2)

B19163-01

Primary Author: Ralph Gordon

Contributor: Steve Muench, Odile Sullivan-Tarazi, Orlando Cordero

Contents

Preface

This manual shows developers how to combine Java 2 Platform, Enterprise Edition (J2EE) and JDeveloper technologies to suit particular application needs. The recommendations in this manual focus on ease of development and recognized best practices that exploit the design-time features of the JDeveloper IDE.

This preface contains the following sections:

- Intended Audience
- Documentation Accessibility
- Related Documents
- Conventions

Intended Audience

This manual is intended for enterprise application developers who want to use JDeveloper to implement enterprise business solutions.

Documentation Accessibility

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

`http://www.oracle.com/accessibility/`

Accessibility of Code Examples in Documentation

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

Accessibility of Links to External Web Sites in Documentation

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

TTY Access to Oracle Support Services

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

Related Documents

For more information:

- For JDeveloper IDE features that support team development, testing, and production deployment to the J2EE platform, see the JDeveloper help system. These topics are beyond the scope of the present document.
- For a list of J2EE-related learning resources, see Section 10.1, "Related Documentation".

Conventions

The following text conventions are used in this document:

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

Part I

Oracle ADF Toy Store Application: Getting Started

Part 1 of the Case Manual contains information to help you set up and become acquainted with the Oracle Toy Store application.

Part 1 contains the following chapters:

1 Introduction to the Oracle ADF Toy Store Application Case Study

This chapter contains the following sections:

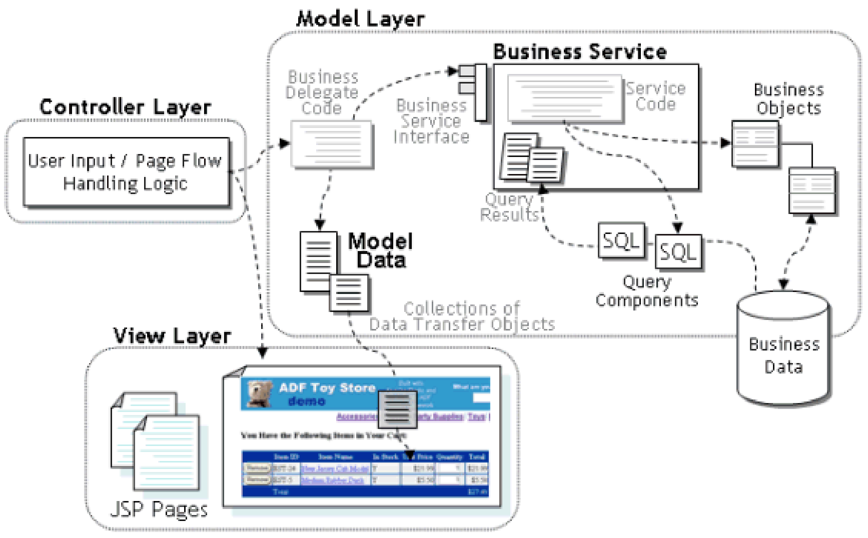
1.1 Introduction

The Oracle ADF Toy Store application is a realistic web store application which consists of over twenty web pages to enable the end user to browse and purchase from a catalog of toys online. Like any web store application, the Oracle ADF Toy Store application includes functionality to manage user accounts, display product details, search the database, manage the shopping cart, and transact purchases. Technically, the application design adheres to the `Model/View/Controller (MVC)` design pattern and is implemented using these existing J2EE application frameworks:

- `Apache Struts`
- `Oracle Application Development Framework (Oracle ADF)`
- `JavaServer Pages (JSP) technology`

As with all MVC-style web applications, Oracle ADF Toy Store has the basic architecture illustrated in Figure 1–1, "The Model-View-Control Architecture".

Figure 1–1 The Model-View-Control Architecture



- The *model layer* represents the business information needed by the application.
- The *controller layer* handles user input, interacts with the model layer, and picks the presentation.
- The *view layer* presents the model data to the end user.

The goal of this case study is to show in detail the framework-based implementation of the Oracle ADF Toy Store sample application. We will focus primarily on the controller and view layer technologies. We will explore the Struts and Oracle ADF frameworks in tandem to understand how they simplify implementing the view and controller layers. In the process, we'll also examine ways to use Oracle JDeveloper 10*g* productively to build these kinds of MVC-style business applications.

1.2 For More Information

Business logic developers responsible for coding the business object layer will want to refer to the Oracle ADF Toy Store technical whitepaper that is a companion document to this case study. The model layer consists of one or more *business services* that expose application functionality and access to *model data* through a business service interface. These business services, in turn, rely on *query components* to retrieve that data and on *business objects* to validate and persist any new or modified data.

In the Oracle ADF Toy Store application, business objects are implemented with Oracle ADF Business Components, a framework within Oracle ADF that supports business logic development. In this case study, we encounter examples of the interaction between the model and controller layers, but an in-depth discussion of the benefits of working with Oracle ADF Business Components is beyond the scope of this study.

To view the technical whitepaper, see the `Welcome to the Oracle ADF Toy Store Application` page on Oracle Technology Network. In particular, the section "Implementing the Model Layer Using Oracle ADF Business Components" will be of interest. The whitepaper provides in-depth descriptions of the benefits of ADF Business Components. The whitepaper also describes in detail parts of the Oracle ADF Toy Store application not addressed by this case study.

Note: In contrast to the companion technical whitepaper, this case study provides detailed descriptions of individual Toy Store application web pages and describes how page flow is managed. We will see exactly how Oracle ADF integrates with the view and controller layers, and how web page designers benefit from Oracle ADF data binding technology and Struts to produce clean, easy-to-understand JSP pages, free of complex business logic and unnecessary scriptlet code.

Before diving into the explanation of the application, let's make sure you can open and run the application in JDeveloper 10*g*. The next chapter details the steps to get the Toy Store application set up correctly on your system.

2 Setting Up the Oracle ADF Toy Store Application

This chapter contains the following sections:

2.1 Introduction

These instructions assume that you are running `Oracle JDeveloper 10g` production, version 10.1.2.0.2. Although the Oracle ADF Toy Store application will work with JDeveloper 10.1.2.0.1 or 9.0.5.*x*, we recommend using the specified production version to follow along with the case study. The application will *not* work with earlier versions of JDeveloper.

We also assume that you have access to an Oracle database, and privileges to create new user accounts to set up the sample data.

Note: Oracle ADF is designed to work with any relational database, and has been tested with Oracle, Oracle Lite, DB2, and SQLServer. The `Using Oracle ADF with Foreign Datasources` whitepaper covers the details. However, for the purposes of this case study we assume that you will use the Oracle database, version 8.1.7 or later.

2.2 Downloading and Extracting the Oracle ADF Toy Store Application

Download the `adftoystore_10_1_2.zip` file if you haven't already.

Extract the contents of the `adftoystore_10_1_2.zip` file with the standard JDK jar utility into a convenient directory:

```
jar -xvf adftoystore_10_1_2.zip
```

The above command will create a directory `adftoystore` and subdirectories. (For these instructions, we assume that you have extracted the `adftoystore_10_1_2.zip` file into the root directory `C:\`, thus creating a "root" directory of `C:\adftoystore`.)

Note: If the `jar` command does not work on your system, double-check that you have included the `<JDKHOME>/bin` subdirectory in your system path. If you downloaded the full version of Oracle JDeveloper 10*g*, then you will have a 1.4.2 JDK in the `<JDEVHOME>/jdk` directory.

2.3 Setting Up the Oracle ADF Toy Store Database Users

Using the SQL script provided, create the `TOYSTORE` and `TOYSTORE_STATEMGMT` user accounts in the database.

Run the create user accounts SQL script like this:

```
cd C:\adftoystore\DatabaseSetup
sqlplus /nolog @CreateToyStoreUsers.sql
```

After you enter your SYS account's password, the script will create the `TOYSTORE` and `TOYSTORE_STATEMGMT` user accounts. The `TOYSTORE` schema will contain the Oracle ADF Toy Store application tables, while the `TOYSTORE_STATEMGMT` schema will be used by the Oracle ADF state management facility to store pending data across web pages.

2.4 Creating the Oracle ADF Toy Store Database Tables

Run the database setup SQL script `./adftoystore/DatabaseSetup/ToyStore.sql` like this:

```
sqlplus toystore/toystore @ToyStore.sql
```

Note: If you have a version of the Oracle database *prior* to Oracle Database 10*g*, the command `purge recyclebin` at the end of this script will yield an error, which you can safely ignore.

2.5 Creating the Oracle JDeveloper Data Connections

In JDeveloper 10*g*, create two database connections to correspond to the two database accounts created above.

Define two connections in JDeveloper 10*g*:

- `toystore`, corresponding to the `TOYSTORE` user (password `TOYSTORE`)
- `toystore_statemgmt`, corresponding to the `TOYSTORE_STATEMGMT` user (password `TOYSTORE`)

Note: The two connection names are *case-sensitive* and should be typed in lowercase, as shown.

To save some typing, you can import these two connections from the supplied `jdev_toystore_connections.xml` file in the `./adftoystore/DatabaseSetup` directory. To do so, select the **Database** category folder in the Connection Navigator and choose **Import Connections** from the context menu. Supply the `jdev_toystore_connections.xml` filename as the file to import from. After importing the two named connections, you should test each connection by selecting it, double-clicking to bring up the Edit Database Connection dialog, and selecting the **Test** tab. If clicking the **Test Connection** button does not yield a "Success!" message, then correct the connection details on the Connection page to work for the database you want to connect to. By default, the connections are defined against a database on your local machine, listening on port `1521`, with a SID of `ORCL`.

2.6 Installing the Oracle JDeveloper JUnit Extension

Optionally, ensure that the JUnit Extension for JDeveloper is installed.

`JUnit` is the standard tool for building regression tests for Java applications. Oracle JDeveloper 10*g* features native support for creating and running JUnit tests, but this feature is installed as a separately downloadable IDE extension. You can tell if you already have the JUnit extension installed by choosing **File > New** from the JDeveloper main menu and verifying that you have a **Unit Tests (JUnit)** category under the **General** top-level category in the New Gallery.

If you do *not* already have the JUnit extension installed, then `download it` now. You'll find it along with all the other extensions available for JDeveloper in the `JDeveloper Extension Exchange` on OTN. To complete the installation of the extension, first exit JDeveloper if you are currently running it. With JDeveloper not running, extract the contents of the downloaded ZIP file into the `./jdev/lib/ext` subdirectory under your JDeveloper installation home directory. Then restart JDeveloper.

Finally, you should verify that the `junit3.8.1` subdirectory exists in your JDeveloper installation home. This directory will be automatically created the first time you run any JUnit wizard from the **Unit Tests (JUnit)** category of the New Gallery. However, even if you are not ready to create any JUnit tests, you may wish to perform the following steps to ensure the directory is set up correctly.

Assuming that your current directory is the JDeveloper installation home directory, run these two commands:

```
jar -xvf jdev/lib/ext/junit_addin.jar junit3.8.1.zip
```

The first command extracts the `junit3.8.1.zip` file from the `junit_addin.jar` archive. The ZIP file contains the distribution of JUnit that JDeveloper has been tested with.

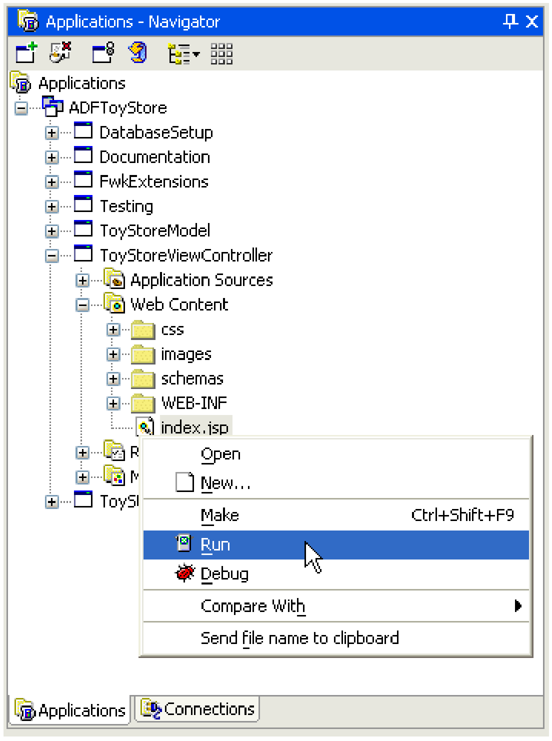
```
jar -xvf junit3.8.1.zip
```

The second command extracts the contents of the junit3.8.1.zip file into the JDeveloper installation home directory.

Open the `./adftoystore/ADFToyStore.jws` workspace in JDeveloper 10*g*.

Run the application in JDeveloper 10g by selecting **index.jsp** in the **ToyStoreViewController** project and choosing **Run**, as shown in Figure 2–1, "Running the Oracle ADF Toy Store Application Inside JDeveloper".

Figure 2–1 Running the Oracle ADF Toy Store Application Inside JDeveloper



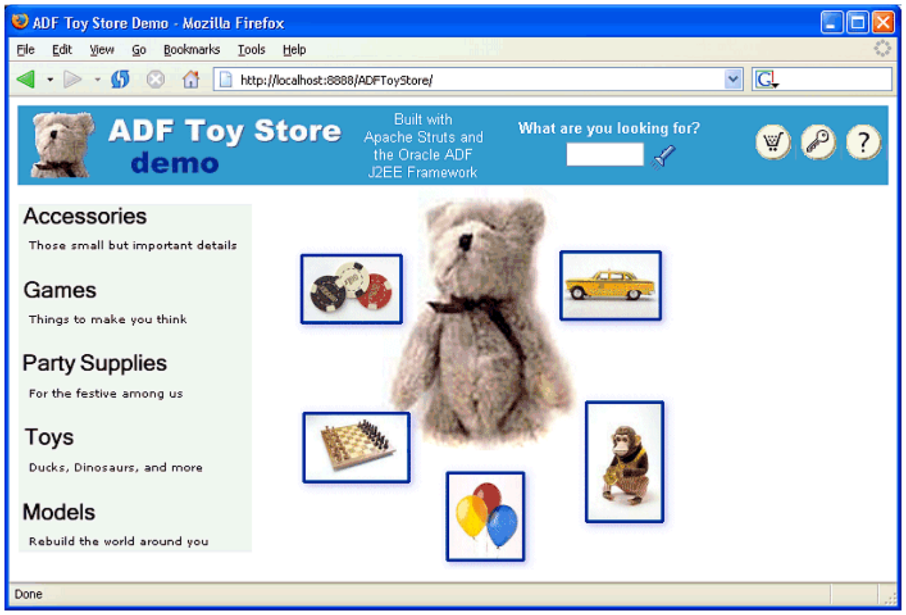
Note: Since `index.jsp` is configured as the default run target on the Runner page of the Project Properties dialog for the ToyStoreViewController project, you can also simply click the **Run** icon in the JDeveloper toolbar when this project is active, or right-click the project and choose **Run**. To see the project's properties, select the project in the navigator, right-click, and choose **Property Properties**.

Running the `index.jsp` page from inside JDeveloper will start the embedded Oracle Application Server 10*g* Oracle Containers for J2EE (OC4J) server, launch your default browser, and cause it to request the following URL:

```
http://yourmachine:8988/ADFToyStore/index.jsp
```

If everything is working correctly, you will see the home page of the Oracle ADF Toy Store application, as shown in Figure 2–2, "Oracle ADF Toy Store Application Home Page".

Figure 2–2 Oracle ADF Toy Store Application Home Page


ADF Toy Store Demo - Mozilla Firefox
File Edit View Go Bookmarks Tools Help
http://localhost:8888/ADFToyStore/
ADF Toy Store
demo
Built with Apache Struts and the Oracle ADF J2EE Framework
What are you looking for?
Accessories
Those small but important details
Games
Things to make you think
Party Supplies
For the festive among us
Toys
Ducks, Dinosaurs, and more
Models
Rebuild the world around you
Done

The next chapter provides a quick tour of the application and helps you to become familiar with the web pages that are the subject of this case study.

3 Quick Tour of the Oracle ADF Toy Store Application

This chapter contains the following sections:

3.1 Introduction

Before examining the individual web pages and their source code in depth, you may find it helpful to become familiar with the functionality of this web store application.

3.2 Browsing Products and Adding Them to Your Cart

The Oracle ADF Toy Store is a realistic online store that sells toys. The products for sale are organized into five categories: accessories, games, party supplies, toys, and models. From the home page, you can browse products in the store in two ways:

- Selecting a category name to see the products in that category

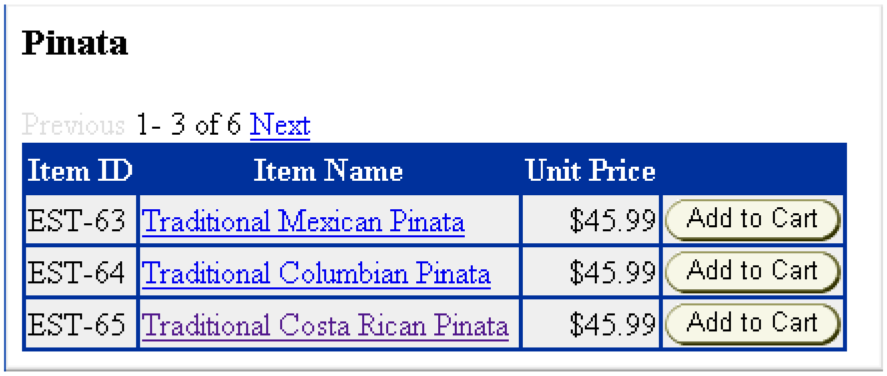
  OR

- Using the **What are you looking for?** search box in the banner to find products by name, regardless of what category they belong to

If the list contains more than three products, products appear a page at a time. You can use the **Next** or **Previous** links above the item list to browse through the complete list.

Click on the name of a product to view a list of the different product items for sale. For example, click **Piñata** to see a list of the different kinds of piñatas that are available, as shown in Figure 3–1, "Browsing a Product Catalog".

Figure 3–1 Browsing a Product Catalog



Pinata

Previous 1- 3 of 6 Next

Item ID	Item Name	Unit Price	
EST-63	Traditional Mexican Pinata	$45.99	Add to Cart
EST-64	Traditional Columbian Pinata	$45.99	Add to Cart
EST-65	Traditional Costa Rican Pinata	$45.99	Add to Cart

On any page where the

Add to Cart

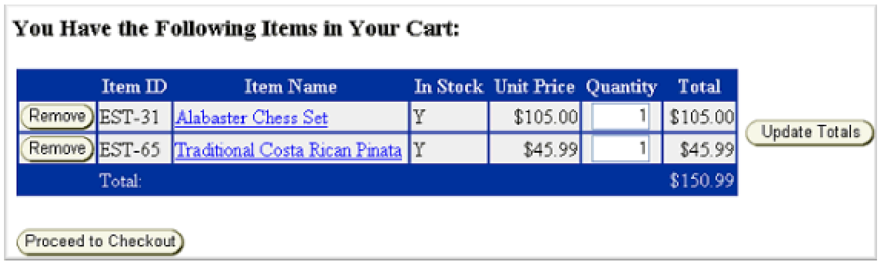
button appears, you can click the button to add one of those items to your shopping cart.

To see a detailed description and a picture of any product, just click on its name.

You can see the items you have in your shopping cart at any time by clicking the

button, which shows a page listing the items and quantities you have selected so far, as shown in Figure 3–2, "Shopping Cart Display".

Figure 3–2 Shopping Cart Display



You Have the Following Items in Your Cart:

	Item ID	Item Name	In Stock	Unit Price	Quantity	Total
Remove	EST-31	Alabaster Chess Set	Y	$105.00	1	$105.00
Remove	EST-65	Traditional Costa Rican Pinata	Y	$45.99	1	$45.99
	Total:					$150.99

Update Totals

Proceed to Checkout

To adjust the quantities of the items in the cart, just type over the current value in the **Quantity** field and click the **Update Totals** button. The recalculated shopping cart total is displayed. You can remove an item from your cart either by clicking the **Remove** button or by adjusting the item to have a zero quantity.

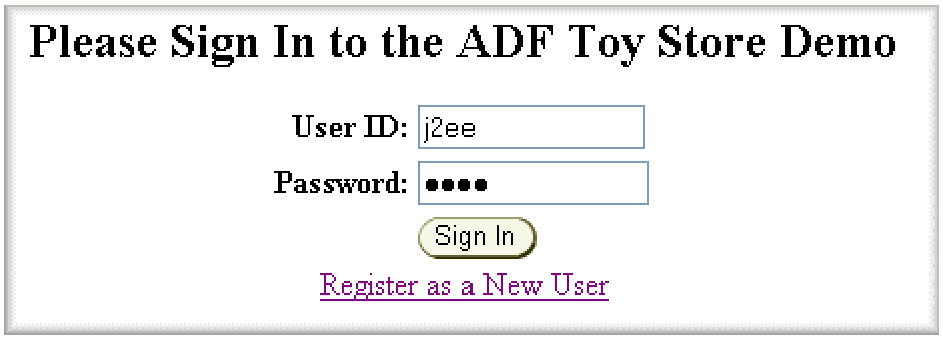
3.3 Checking Out and Signing In

From the Shopping Cart page, click the **Proceed to Checkout** button to proceed to the Review Checkout page. From there, you can review your purchase and, if you are happy with it, click the Continue button to continue.

If you have not already signed into the web store site as a registered user, you will be prompted to sign in at this point to continue with the checkout process. Your sign-in page will resemble Figure 3–3, "Sign-In Page". The user named `j2ee` is already registered, with a password of `j2ee`, so you can provide these credentials to continue.

Note: If instead you want to register as a new user, click the **Register as New User** link. See the next section for details.

Figure 3–3 Sign-In Page



Please Sign In to the ADF Toy Store Demo

User ID: j2ee

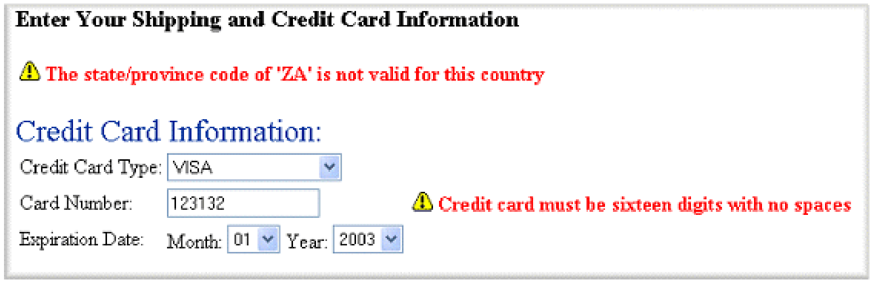
Password: ●●●●

After you successfully sign in, proceed to the page where you can confirm your shipping and payment details. This part of the appliation lets you try out some of the application validation logic that has been implemented. You can try:

- Entering an invalid state abbreviation of ZA for the country USA
- Entering a credit card number that does not comprise 16 digits

and clicking the **Continue** button. You should see multiple validation errors, as shown in Figure 3–4, "Validation Errors on Submitted Form".

Figure 3–4 Validation Errors on Submitted Form



Enter Your Shipping and Credit Card Information

The state/province code of 'ZA' is not valid for this country

Credit Card Information:

Credit Card Type: VISA

Card Number: 123132 Credit card must be sixteen digits with no spaces

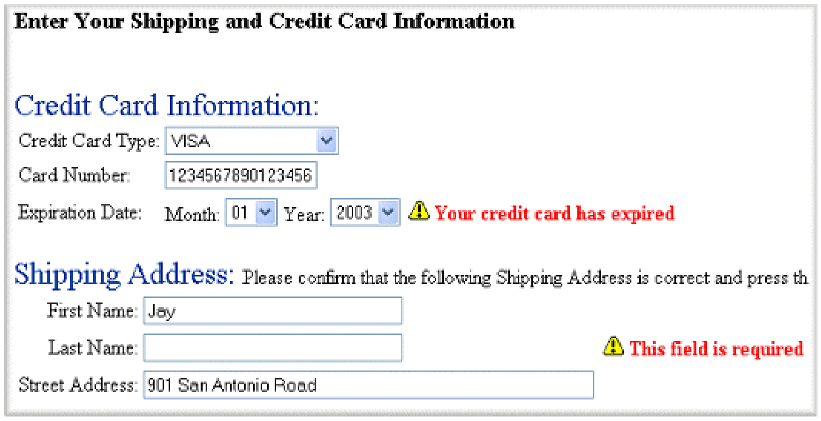
Expiration Date: Month: 01 Year: 2003

After fixing those errors by entering a valid state abbreviation like CA, and filling out a full 16-digit credit card number, try causing some additional validation errors by:

- Entering a date in the past for the expiration date of your credit card
- Blanking out a required field like **Last Name**

When you click the **Continue** button again, you will see the validation errors that need to be corrected, as shown in Figure 3–5, "Additional Shipping Validation Error Information".

Figure 3–5 Additional Shipping Validation Error Information


Enter Your Shipping and Credit Card Information

Credit Card Information:

Credit Card Type:	VISA
Card Number:	1234567890123456
Expiration Date:	Month: 01 Year: 2003 ⚠ Your credit card has expired

Shipping Address: Please confirm that the following Shipping Address is correct and press th

First Name:	Jey	
Last Name:		⚠ This field is required
Street Address:	901 San Antonio Road	

After you correct these final validation problems and resubmit your order, the order will be placed, and you'll see the final Thank You page, with an order reference number listed. Clicking on the hyperlinked order reference number takes you to an order summary page, which is implemented using the XML/XSLT-based Oracle XSQL Pages publishing framework instead of JSP pages, to illustrate that multiple view-rendering technologies are possible. (Note that this Order Summary web page is not described in this case study but is covered by the Oracle ADF Toy Store technical whitepaper.)

3.4 Registering a New User and Editing an Existing User's Profile

If you are not currently logged in as a registered user of the web store, you can create a new account by clicking the

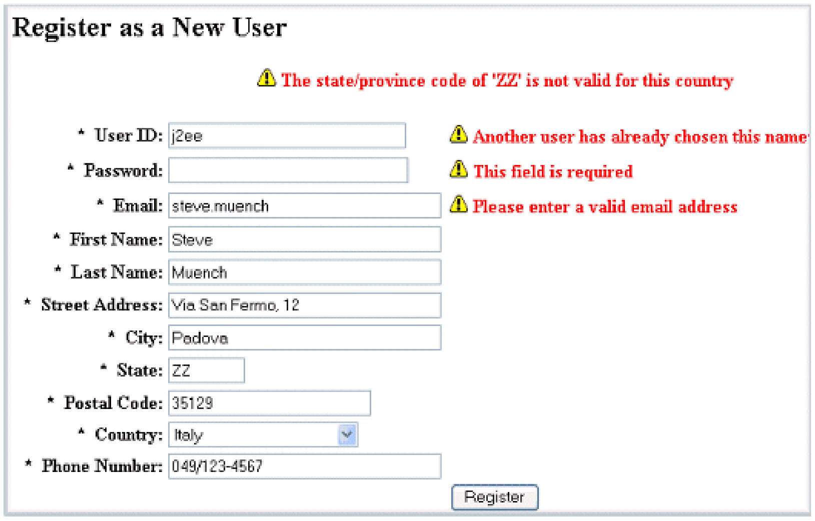
button, which brings you to the Sign-In page, as shown in Figure 3–3, "Sign-In Page". From there, you can register as a new user by clicking on the **Register as a New User** link. This link takes you to a form where you can enter the necessary registration details.

This user registration page is another place in the application where it's easy to observe how business rules are enforced by the Oracle ADF framework. For example, if while filling out the form you:

- Enter a user name that has already been chosen by another user
- Forget to provide a password
- Enter an email address that is not properly formed

then when you submit the form, you'll see the full set of errors related to your registration, as shown in Figure 3–6, "New User Registration Validation Information".

Figure 3–6 New User Registration Validation Information



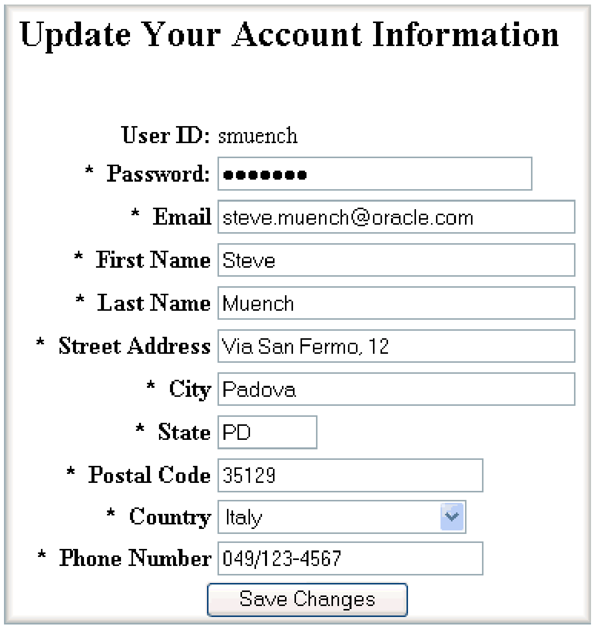
Register as a New User

⚠ The state/province code of 'ZZ' is not valid for this country

* User ID:	j2ee	⚠ Another user has already chosen this name
* Password:		⚠ This field is required
* Email:	steve.muench	⚠ Please enter a valid email address
* First Name:	Steve	
* Last Name:	Muench	
* Street Address:	Via San Fermo, 12	
* City:	Padova	
* State:	ZZ	
* Postal Code:	35129	
* Country:	Italy	
* Phone Number:	049/123-4567	

Of course, since we're working with the same underlying business object that represents user accounts here in this Update Account form, the same validation will be enforced as with the new user account. Figure 3–7, "Editing Account Details", shows the information that is to be validated when customers update their accounts.

Figure 3–7 Editing Account Details



Update Your Account Information

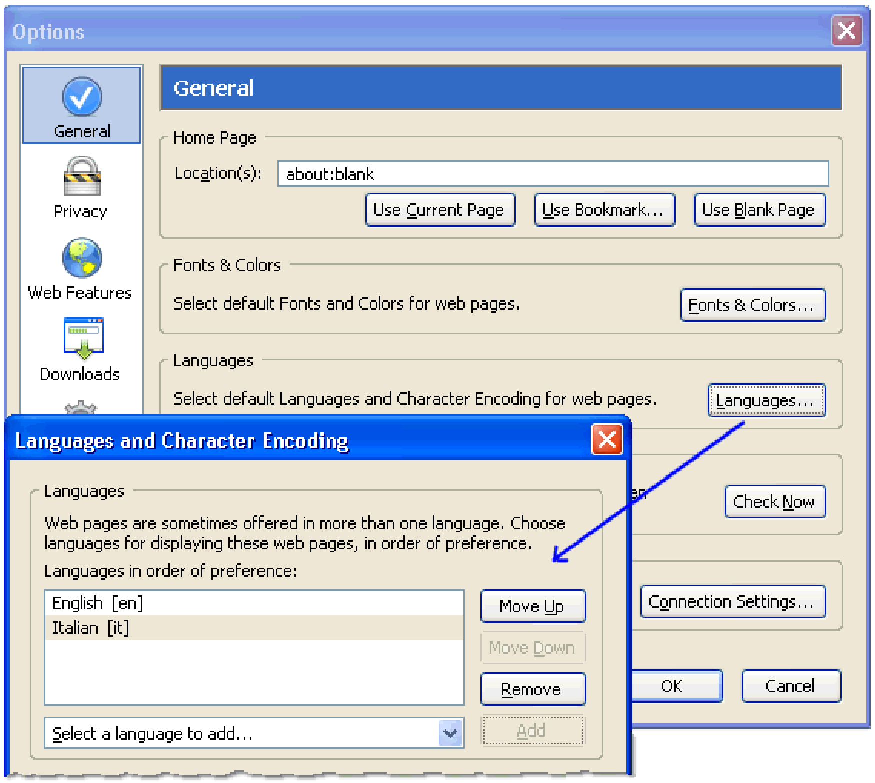
User ID:	smuench
* Password:	●●●●●●●
* Email	steve.muench@oracle.com
* First Name	Steve
* Last Name	Muench
* Street Address	Via San Fermo, 12
* City	Padova
* State	PD
* Postal Code	35129
* Country	Italy
* Phone Number	049/123-4567

3.5 Trying Out the Toy Store Web Application in Another Language

The application is built using the internationalization features supported by Struts and Oracle ADF Business Components, and it ships with support for three languages: English (the default), Italian, and German. The Struts and Oracle ADF frameworks determine the language to be displayed, based on your browser settings. You can see what the web pages look like in Italian by setting your browser language preferences appropriately.

In Mozilla Firefox 0.9, you select your preferred languages on the General page of the **Tools > Options** menu by clicking the **Languages** button, as shown in Figure 3–8, "Setting Preferred Languages in Mozilla Firefox". You can add **Italian [it]** and then click **Move Up** to move it to the top of the list.

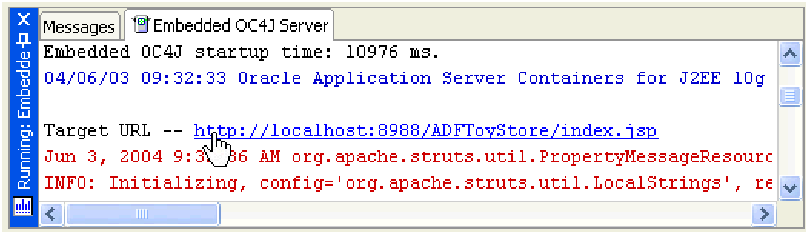
Figure 3–8 Setting Preferred Languages in Mozilla Firefox


Options
General
Privacy
Web Features
Downloads
General
Home Page
Location(s): about:blank
Use Current Page
Use Bookmark...
Use Blank Page
Fonts & Colors
Select default Fonts and Colors for web pages.
Fonts & Colors...
Languages
Select default Languages and Character Encoding for web pages.
Languages...
Check Now
Connection Settings...
OK
Cancel
Languages and Character Encoding
Languages
Web pages are sometimes offered in more than one language. Choose languages for displaying these web pages, in order of preference.
Languages in order of preference:
English [en]
Italian [it]
Move Up
Move Down
Remove
Select a language to add...
Add

In Internet Explorer, you can do the same thing by choosing **Tools > Internet Options** and by clicking on the **Languages** button in the General page of the dialog.

Since Apache Struts caches the browser user's preferred language at the servlet session level, you will need to close the current browser window and open a new one before you'll see the application change to Italian. A quick way to relaunch your preferred browser with the right URL for the application is to find and click the target URL in the JDeveloper 10*g* Log window, as shown in Figure 3–9, "Relaunching the Browser Window".

Figure 3–9 Relaunching the Browser Window

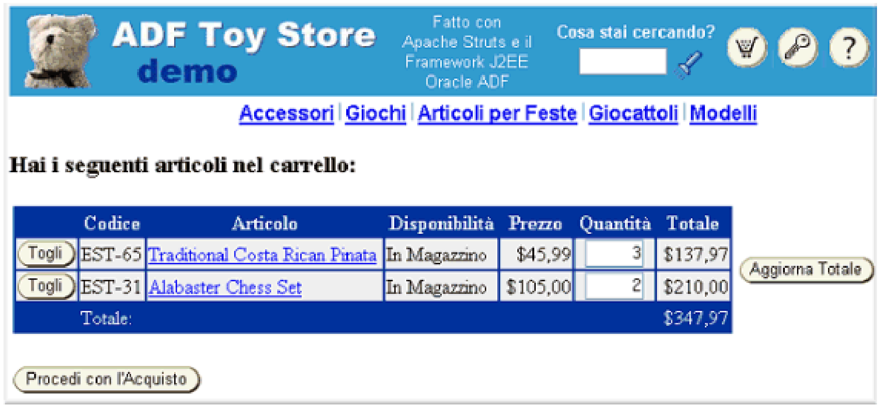


```
Embedded OC4J startup time: 10976 ms.
04/06/03 09:32:33 Oracle Application Server Containers for J2EE 10g

Target URL -- http://localhost:8988/ADFToyStore/index.jsp
Jun 3, 2004 9:3  36 AM org.apache.struts.util.PropertyMessageResourc
INFO: Initializing, config='org.apache.struts.util.LocalStrings', re
```

Clicking on the Log URL will start the Oracle ADF Toy Store application again, but this time in Italian. After you add the same items to your shopping cart, your page will resemble what you see in Figure 3–10, "The Shopping Cart Home Page in Italian".

Figure 3–10 The Shopping Cart Home Page in Italian


ADF Toy Store
demo
Fatto con Apache Struts e il Framework J2EE Oracle ADF
Cosa stai cercando?
Accessori | Giochi | Articoli per Feste | Giocattoli | Modelli

Hai i seguenti articoli nel carrello:

	Codice	Articolo	Disponibilità	Prezzo	Quantità	Totale
Togli	EST-65	Traditional Costa Rican Pinata	In Magazzino	$45,99	3	$137,97
Togli	EST-31	Alabaster Chess Set	In Magazzino	$105,00	2	$210,00
	Totale:					$347,97

Aggiorna Totale
Procedi con l'Acquisto

You can set your browser's preferred language back to English, and close and relaunch the browser window to proceed in English again.

The next chapter begins the first lesson of the case study, where you will investigate designing databound JSP pages in JDeveloper.

Part II

The Oracle ADF Toy Store Web Application: Lessons

Part 2 of the Case Manual decomposes the Struts page flow of the Oracle ADF Toy Store application. Each chapter explores the concepts of Oracle ADF as they would be employed in a typical web store application and concludes with a hands-on tutorial that you may complete.

Part 2 contains the following chapters:

4 Lesson One: Designing the Home Page

This chapter contains the following sections:

4.1 Introduction

The home page represents our store front. It is the first thing the user sees when entering our Toy Store site. The home page must both attract potential customers and serve returning customers. To accomplish these aims, the web store home page typically:

- Allows the customer to browse products within specific categories (ours are Accessories, Games, Party Supplies, and so on)
- Allows the customer to quickly locate specific products by typing keywords in a search field
- Allows the customer to view and edit the contents of their shopping cart
- And, of course, allows customers to register/sign in and edit their personal account information

4.2 Planning the Design of the Home Page

Our web store home page is a good place to begin planning our application. In JDeveloper, it is tempting to open the HTML/JSP Visual Editor and begin laying out the pages. But initially we need to address some basic questions about the underlying technology:

- What model layer objects will the home page require?
- Do some of the model layer objects represent common operations that we can access across the application's pages?
- Do we anticipate localizing the application?

In the initial planning phase, it makes sense to focus on the home page. While the home page of our web store application does not represent 100 percent of the required application business objects, we can identify important objects, including the

`ProductList`, `Accounts`, and `ShoppingCart` objects. Although the finished application will utilize additional business objects, creating the model layer can be iterative, meaning that we can expose those business objects as we require them.

Next, the home page forces us to think about the controller layer, because we can begin to see how the page flow is organized. With Struts as our controller layer, we'll identify "action forwards" like `showcategory.do` and `search.do` to specify targets for the home page links and operations.

To invite users to browse the store, we want our application to have a well-considered site navigation. For example, we don't want users to have to return to the home page just to log in, view their cart, and search products. Instead, these operations should appear across the application. Our first page design decision springs from the need to reuse these home page operations: we will create a `header.jsp` banner with the common operations displayed and utilize the JSP include directive to insert the banner into the desired pages. This best practice allows us to reduce the number of pages and simplify site navigation.

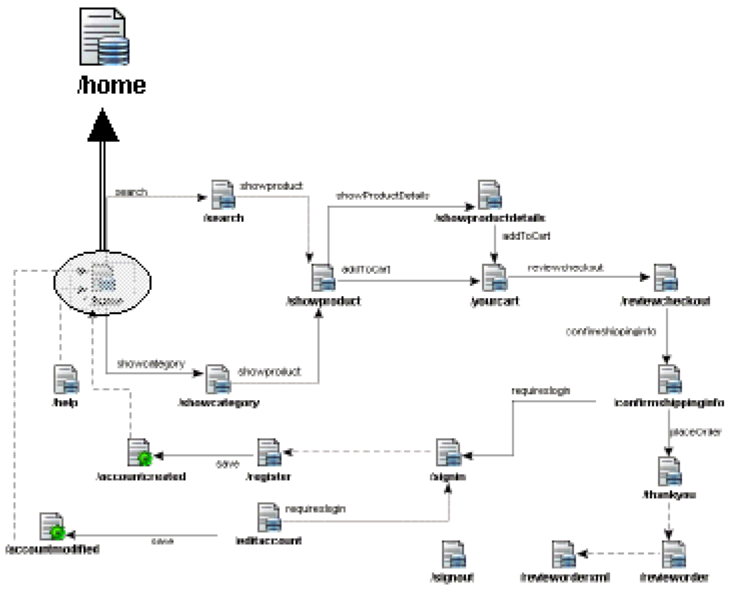
Finally, since we are committed to hosting our site internationally, we will utilize the Struts Bean tags to retreive translated strings from locale-specific message resource files that we will create. In the hands-on for this lesson, we'll see exactly how easy it is to utilize this capablity of Struts in JDeveloper.

4.3 Getting Started with the Struts Page Flow Diagram

The Struts page flow diagram in JDeveloper is your workbench for creating pages, accessing the page action's implementing action class (or Oracle ADF data action classes, in some cases), visualizing the action forwards and page links of the page flow, running the application by initiating a Struts action `.do` request (we never run JSP pages directly in a Struts-based web application), and even creating and opening HTML or JSP pages for editing in the visual editor. That's a lot of activities to be aware of, but through the course of this project, we'll utilize them all.

Each lesson focuses on a small portion of the complete page flow diagram. In Lesson 1, we limit the discussion to the component labeled `/home`. Figure 4–1, "The Home Data Page Icon Represents the Postback Pattern", illustrates this component’s position in the overall page flow.

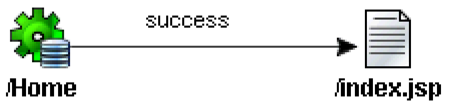
Figure 4–1 The Home Data Page Icon Represents the Postback Pattern



Absent from the page flow diagram are any JSP pages. This is because our JSP pages are databound and therefore represent a combination of components. The icon shown for the home page is typical of the majority of the Struts page flow components: it is the symbol for a data page and is available from the Struts Page Flow Component Palette.

Each data page in the page flow represents Oracle ADF's integration with Struts action classes. For developing page flows with databound JSP pages, the data page is very convenient because it combines a Struts action class (`DataForwardAction` or subclass), a Struts action forward, and a target web page in a single icon. Alternatively, the page flow could display several icons to represent the postback. Figure 4–2, "The Postback Pattern Implemented Without a Data Page", depicts the same postback as it would appear in the page flow diagram using the Oracle ADF data action (labeled **/Home**), a **success** forward, and, finally, the target **index.jsp** page.

Figure 4–2 The Postback Pattern Implemented Without a Data Page



The single data page icon effectively represents the typical use case where a web page posts back to the action that initially set up the page's data. The postback pattern allows the action to handle page events before initiating the page flow action forward. The underlying Oracle ADF element, data action, is also available on the Component Palette and may be used in a page flow when finer granularity is required or when the postback pattern is not required.

As we'll see in the next lesson, the page flow developer implements a data action (corresponding to the data page visualized in the page flow) to perform specific work on the model and to customize the Oracle ADF request-handling lifecycle. Because the Oracle ADF lifecycle prepares the Oracle ADF binding context for databound web pages, it is specifically this integration between Struts and Oracle ADF that provides the glue between the presentation layer and the model layer where the data lies.

4.4 Laying Out the Home Page

With a minimum of information, including a useful page name, a starting page flow, and a nice way to include common operations into our page, we are ready to begin.

In the Struts page flow diagram, double-click **/home** to open the `index.jsp` page in the visual editor design view. Overall the page design is very clean with a minimum of tags. The include directive appears at the top of the page to include our reusable page. Below that appears the body of our home page: a two-column table element, with another table embedded in the first column to provide the product category links. We also see the message tag displayed with the key to reference in the application resource `.properties` file. The message tag, available from the Component Palette Struts Bean list, serves our internationalization requirement.

Note: The home page follows the naming convention, `index.jsp`, which means that site users don't need to type the full home page URL. For example, the web server would automatically redirect the URL `http://www.toystore.com` to the application start page at `http://www.toystore.com/index.html`. Your web server configuration file determines the actual default start page names. In the Oracle ADF Toy Store application, we use a second `index.jsp` page at the *root* of `WEB-INF` to trigger a corresponding Struts action (through a `<jsp:forward>` tag to the `home.do` action). The Struts controller then executes a data page related to the actual databound page `/WEB-INF/jsp/index.jsp`.

Notice that there are no text links displayed in the page. This is because the home pages utilizes images as links. It is worth noting that we utilize the URL rewriting capability of the `JSTL <c:url>` tag to construct the URLs for each link in the home page. The `<c:url>` tag builds the URL by adding a session ID (for cases where browser cookies are disabled) to maintain session information as the user navigates the site. In this example, the fully encoded URL is then used as the href attribute value in the HTML link element:

```
<a id="accessories" href="<c:url value='showcategory.do&id=A'/>">
    <img src="<bean:message key='images.sidebar.A'/>" border="0"/>
</a>
```

We will make use of the above constuction throughout the application pages. The `<c:url>` tag's value attribute specifies the Struts action to invoke and passes the `id` argument to the associated action handler class. We will examine the action handler in the next lesson.

Troubleshooting Tip: When you run your web application in a browser with cookies disabled, the error `javax.servlet.jsp.JspException: Missing message for key "XYZ"` may appear when you click a link. If you want to allow anonymous users to browse your application, you must use the URL rewriting capability of the `<c:url>` tag when constructing your application page's links. When executed, the tag rewrites the URL with the `JSESSIONID` cookie in the page.

Complete the following hands-on task before proceeding to the next lesson if you would like to explore concepts described in this lesson.

Note: Completing a hands-on task is optional. If you choose not to complete a particular hands-on, you can still continue with subsequent lessons and hands-on tasks.

4.5 Hands-On for Lesson 1

The following hands-on shows how to quickly create and use a new translatable message resource within any JSP page.

1. In the Application Navigator, expand **toystore.view** in **ToyStoreViewController** and double-click **ToyStoreResources.properties**.
2. In the open file, locate the `index.P` resource string and insert the text:

   ```
   index.Z=The high-tech variety
   ```
3. Close the file and save the new resource with it.
4. Open the `index.jsp` home page in the design view of the visual editor.
5. Right-click inside the last table row of the single-column table (displays message index M) and choose **Table-Insert Rows or Columns**.
6. In the dialog, make the necessary selections to insert a single new row below the current row.
7. Click in the newly inserted row and display **Struts Bean** tags in the Component Palette.
8. Click the message tag from the Component Palette to insert the tag into the empty table cell.
9. To display the tag editor, double-click the inserted tag icon in the open page. Scroll the editor to locate the key attribute, click the dropdown icon, and choose **index.Z** from the list. The icon will be updated with the key name.
10. From the Struts page flow diagram, right-click the **home.do** action and choose **Run** to launch the application. Your new page will display the newly created message resource.

You may want to return to JDeveloper to copy the full link text from one of the table cells and paste it in the same table cell as the newly created message resource. Because the link is constructed with embedded JSTL tags within the HTML link tag, it is not possible to work exclusively with the Component Palette to build the link. In this case, we recommend using the source editor to modify the source directly.

The next lesson describes how to expand the Struts page flow with additional databound JSP pages.

5 Lesson Two: Drilling Down Into the Products

This chapter contains the following sections:

5.1 Introduction

We began our decomposition of the Oracle ADF Toy Store web store application with its home page. In Lesson 1, we decided users should be able to immediately initiate product searches and begin browsing the catalog without needing to log in or accept application cookies. Now we are ready to create JSP pages to support the product search and category display home page operations. To accomplish this, we will need to perform several tasks:

- Extend the Struts page flow to display products of interest and allow the user to drill down for individual product details.
- Ensure that the business model supports drilling down to display the product details.
- Look at common design patterns for displaying categories, products, and product details.

Let's begin by returning to the application workbench, the Struts page flow diagram, to extend our page flow based on these requirements.

5.2 Analyzing the Products Display Page Flow

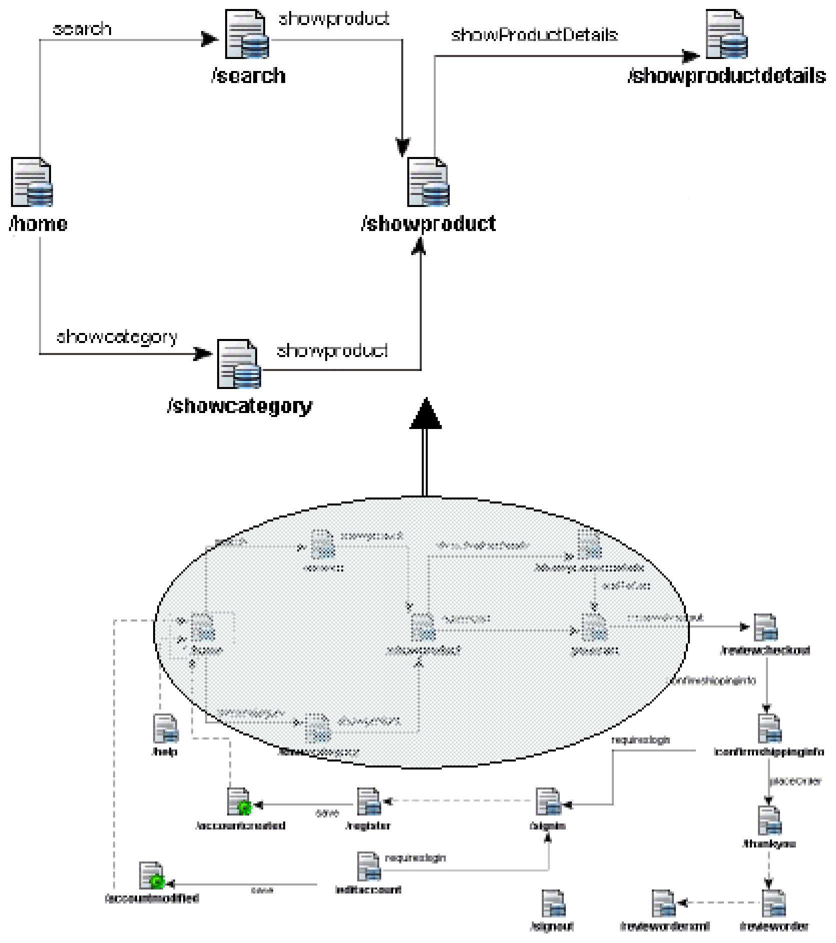
The page flow from the home page contains two branches: `search` and `showcategory`. As we would expect, separate action forwards correspond to separate JSP pages:

- The `search.jsp` page as the target of the `search` forward must iterate over matching products and display feedback when no product name matches the supplied query string.

- The `category.jsp` page as the target of the `showcategory` forward needs only to iterate over the list of products that match the supplied category `id`.

Following these two pages, we'll want to display the drilldown page where the user can view actual product details before deciding whether to update their cart with a product selection. The new page flow, with the drilldown data page **/showproductdetail**, is shown in Figure 5–1, "Product Display Page Flow".

Figure 5–1 Product Display Page Flow



We continue to use the Oracle ADF data page construct to post back to data actions from mapped JSP pages. The data page's underlying data action ensures that the Oracle ADF model layer is updated before the controller displays the target databound JSP page.

5.3 Storing JSP Files Under the WEB-INF Directory

To protect your JSP pages from the user typing a URL to directly access or view any page, store your pages in a subfolder below the web application's `WEB-INF` directory. Based on the servlet specification, `WEB-INF` is not part of the public document tree of the web application. Therefore, no resource within the `WEB-INF` directory, or its subfolders, may be served directly to a client. If you were to save the files under the public tree, at the web application root level, the user could bypass the Struts controller by invoking a JSP directly, which would mean being served pages with no data binding.

Once the JSPs appear below the `WEB-INF` directory, you must use "`WEB-INF`" as part of the URL when referencing the pages. For example, in our Struts configuration file, the action mapping for `showcategory` specifies the `showcategory.jsp` parameter by its complete path:

```
<action path="/showcategory" ...attributes not shown....
        parameter="/WEB-INF/jsp/showcategory.jsp" unknown="false">
```

To take advantage of hiding the JSP pages under `WEB-INF`, you must always invoke your JSP pages with a Struts `.do` action request, even if the action is a very basic JSP. Additionally, be aware when storing your JSP pages below `WEB-INF` that not all web containers support this feature. Be sure to check your specific container.

5.4 Integrating the Model and Controller Layers

We already know that different pages will be required to handle the search, category list, and product details, and we know that the query parameters used to find matches are different (product names for searches, and category-unique IDs for category and product links), but underlying all three pages is the need to prepare a query object with the supplied bind parameter and to iterate over the result set. Iterating over the result set is something we'll look at in the page descriptions below. However, in an MVC web application that uses Struts and Oracle ADF to access the model layer, we separate the code needed to prepare the query and create a result set from the presentation layer.

The task of accessing the model layer is initiated in the Oracle ADF data action class, which we described in Lesson 1. Thus, the presentation layer need not have any awareness of how the data is accessed. The data action gets executed like any other Struts action class, except that it provides methods you can use to operate on the model layer. Take a look at these action mappings from the Struts configuration file:

```
<action-mappings>
   <action path="/showcategory"
            className="oracle.adf.controller.struts.actions.DataActionMapping"
            type="toystore.controller.strutsactions.ShowCategoryAction"
            name="DataForm" parameter="/WEB-INF/jsp/showcategory.jsp"
            unknown="false">
            <set-property property="modelReference"
                          value="WEB_INF_jsp_showcategoryUIModel"/>
            <forward name="showproduct" path="/showproduct.do" />
   </action>
   <action path="/showproduct"
            className="oracle.adf.controller.struts.actions.DataActionMapping"
            type="toystore.controller.strutsactions.ShowProductAction"
            name="DataForm" parameter="/WEB-INF/jsp/showproduct.jsp"
            unknown="false">
            <set-property property="modelReference"
                          value="WEB_INF_jsp_showproductUIModel"/>
            <forward name="addToCart" path="/yourcart.do" />
            <forward name="showProductDetails" path="/showproductdetails.do"/>
    </action>
...
    <action path="/showproductdetails"
            className="oracle.adf.controller.struts.actions.DataActionMapping"
            type="toystore.controller.strutsactions.ShowProductDetailsAction"
            name="DataForm" parameter="/WEB-INF/jsp/showproductdetails.jsp"
            unknown="false">
            <set-property property="modelReference"
            value="WEB_INF_jsp_ showproductdetailsUIModel"/>
```

```
                <forward name="addToCart" path="/yourcart.do"/>
    </action>
...
</action-mappings>
```

The `type` attribute informs each of our three actions about which class will be executed on the action request. To view the implementation of the data action, return to the page flow diagram, right-click the **/showcategory** data page and choose **Go to Code**. Our action classes, like the one shown in the source editor, allow the application to do some specific work before the Oracle ADF lifecycle prepares the data bindings for the model layer. In our case, we want to create a query object by setting bind variables passed from the invoking JSP page.

We come upon another important design decision at this stage: we can either implement logic to set up the queries on business object data within each data action, or we can invoke service method on our business objects. The latter approach is recommended as a best practice, because it allows the business objects to encapsulate the implementation details of the service, it keeps the controller layer very thin, and it permits the developer to create simple JUnit tests to test the business objects. The data action only implements a custom method `initializeModelForPage()` to retrieve the user-supplied parameters from the `HttpServletRequest`. This custom method, in turn, passes the parameter as an argument to the `prepareToShowProductDetails()` method on our `ToyStoreService` business service interface. The `prepareToShow`*`Xxx`*`()` service methods, in turn, execute the query with the correct binding values before the Struts controller serves the JSP page.

Troubleshooting Tip: Bind variables must be set before the Oracle ADF lifecycle's `prepareModel()` phase is initiated. Otherwise, exception `JBO-27122: SQL error during statement preparation` will be caused, because the query will not be completed. To avoid this error for this common use case (setting bind variables supplied by the user), it is recommended that you override the `prepareModel()` phase of the Oracle ADF lifecycle to include the custom method `initializeModelForPage()`. For an extended discussion of how to customize the Oracle ADF lifecycle for this purpose, see the section "A Page Showing Results of a Query with Bind Variables" in the `Building a Web Store with Apache Struts and Oracle ADF Frameworks` whitepaper.

Best Practice Tip: Business services developers will want to investigate further the benefits of exposing service methods on the application's business objects. By defining these methods outside of Struts and the web container context, it is possible to take advantage of the JUnit testing framework support in JDeveloper (available as a small, separate download, as explained in Section 2.6, "Installing the Oracle JDeveloper JUnit Extension"). One of the wizards available in the **Unit Tests (JUnit)** category of the New Gallery allows you to create a skeleton Business Components test suite. This wizard allows you to pick an application module component, and a particular configuration that you'd like to use for testing, and then to generate:

- A JUnit test fixture that encapsulates getting an instance of the desired application module
- A sample JUnit test case class which uses this fixture and asserts that all expected view object instances exist in the application module's model data map
- A JUnit test class that runs all of the test cases (initially just one)

The above wizard and the built-in JDeveloper refactoring tools were used to create the Oracle ADF Toy Store application packages containing the test runner, the unit test cases, and the test fixture as separate packages. These packages were created to exercise the `toystore.model.services.ToyStoreService` application module component. In the JDeveloper Application Navigator, expand the **Testing** project node and the **ADFToyStore** application node to view the supplied test suite.

5.5 Laying Out the Query Results Page

At runtime, the action mapping for the home page action specifies one of two outcomes: a `/search` data page (`search.do`) or a `/showcategory` data page (`showcategories.do`). As has already been described, the data action for each prepares the model before the web page is displayed in the browser.

The task of rendering the databound page is left to the four tag libraries:

- The Struts Bean tag library, whose tags are prefixed by `bean:`
- The Struts HTML tag library, whose tags are prefixed by `html:`
- The JSTL Core tag library, whose tags are prefixed by `c:`
- The Oracle ADF tag library, whose tags are prefixed by `adf:`

Again, we use the Struts Bean tag library `<bean:message>` tag to include translatable text strings into our pages, based on string keys like `category.productid`, `category.productname`, or the scriptlet-derived `<%= request.getParameter("id") %>`, where `id` corresponds to the category name set up by the URL selection in the home page.

We see the `<html:errors>` tag from the Struts HTML tag library, which makes it easy to display any errors that occur during runtime processing at the top of the page in a standard way.

However, the real work of pulling the data from the bindings in the Oracle ADF model layer is done with the aid of the JSTL Core tag library. The HTML table element provides the formatting for the displayed rows of data generated by the JSTL tags. This common presentation layer use case looks like this in the `showcategory.jsp` page:

```
<table id="categorydata" border="0" bgcolor="#003399">
   <tr>
     <th><font color="white" size="3"><bean:message key="category.productid"/>
         </font>
     </th>
     <th><font color="white" size="3"><bean:message key="category.productname"/>
         </font>
     </th>
   </tr>
   <c:forEach var="Row" items="${bindings.ProductsInCategory.rangeSet}" >
      <tr bgcolor="#f3f3f3">
         <td><c:out value="${Row.Productid}" /></td>
         <td>
            <a href="<c:url
                 value='showcategory.do?event=showproduct&id=${Row.Productid}'/>">
               <c:out value="${Row.Name}" />
            </a>
         </td>
      </tr>
   </c:forEach>
</table>
```

Specifically, the `<c:forEach>` tag iterates over our data collections and the `<c:out>` tags display values for their attributes in the page. Embedded within the `<c:out>` tags is the first use of tag attribute expressions to access objects in the Oracle ADF data binding context. These expressions begin with a variable "`Row`", which is defined by the evaluated expression `${bindings.ProductsInCategory.rangeset}`. The namespace identifier `bindings` locates the Oracle ADF binding context and its objects, `ProductsInCategory` is the name of the table bindings, and `rangeSet` refers to a property of the table binding which supplies access to the individual rows of the bound business object.

To review, Oracle ADF bindings are lightweight objects that decouple back-end data and front-end UI display:

- An iterator binding lets your pages work with a collection of business objects supplied by the Oracle ADF model layer.

  In the `showcategory.jsp` page, the iterator `ProductsInCategoryIterator` serves this purpose. The iterator binding itself is not referenced in the page, but is part of the definition for the control bindings specified in the `UIModel.xml` file for the page.

- Control bindings provide a standard interface for display components in the page to interact with an iterator's data or to invoke "action" methods for preparing model data and handling events.

  In the `showcategory.jsp` page, we see an Oracle ADF table binding referenced by `bindings.ProductsInCategory`, which defines which attributes to display. The binding container for this JSP page also contains two action bindings [`next` and `previous` (but, as we'll see, these are actually invoked through a JSP page we will include [`pagingControl.jsp`).

The important point to consider is that this common presentation layer use case (iterating over your data and formatting the output using standard tags) is done without scriplet code to manipulate the model layer directly in the JSP page. For example, in the Oracle ADF Toy Store application, using just JSTL tags and HTML elements, we see the same pattern repeated in the `search.jsp`, `showcategory.jsp`, and `showproduct.jsp` pages.

Let's examine another example of how Oracle ADF control bindings access the model layer. Open the `showproductdetails.jsp` page in the JDeveloper source editor. In the source for that page, notice that we do not use the previously described Oracle ADF table binding and `<c:forEach>` loop because the HTML table displays only a single row (information for a single product). Instead, the page source contains a series of `<c:out>` tags and attribute expressions that directly specify the namespace identifier bindings to locate the Oracle ADF binding context and its objects:

```
...
<c:out value="${bindings.Name}"/>
...
```

The reference following `bindings` in these expressions is the name of the Oracle ADF attribute binding which identifies the bound attribute of the business object. Even without the `<c:forEach>` loop and the `Row` variable assignment previously seen in the source for `showcategory.jsp`, the attribute bindings of `showproductdetails.jsp` can still obtain the attribute value corresponding to the current row of the underlying business object. This works in `showproductdetails.jsp` because control bindings are always bound to the current row of the business object through an iterator binding. So, retrieving the values on these attribute bindings will automatically retrieve the values on the business object row with the current focus. In this way, coordination between the presentation layer and the model layer is managed for you through the Oracle ADF iterator and control bindings.

In the hands-on for this lesson, we'll see exactly how easy it is to add databound controls to your JSP in JDeveloper.

5.6 Conditionalizing the Display

The page `search.jsp` relies on another set of tags from the JSTL Core tag library worth examining: `<c:choose>`, `<c:when>`, and `<c:otherwise>`. The search page uses these tags like an if / then / else statement to conditionalize the display based on whether the search result contains matching products or not. Here is an excerpt from `search.jsp`:

```
<c:choose>
  <c:when test="${not empty bindings.FindProducts.rangeSet}">
    <table border="0" bgcolor="#003399" >
       <tr>
          <!-- displays table header columns here  -->
       </tr>
       <c:forEach var="row" items="${bindings.FindProducts.rangeSet}" >
          <tr>
             <!-- displays table body here  -->
          </tr>
       </c:forEach>
    </table>
   </c:when>
   <c:otherwise>
     <br><br><bean:message key="search.nomatchingproducts"/>
   </c:otherwise>
</c:choose>
```

You may notice the `<adf:render>` tag from the Oracle ADF tag library (omitted in the above excerpt), which appears in the third column definition of the product items table. This tag displays and formats attribute data based on language-sensitive format masks that you can define in your Oracle ADF Business Components project.

Troubleshooting Tip: Each Oracle ADF Toy Store web page includes an `<html:errors>` tag at the top. By default, the Oracle ADF data action bundles up all exceptions that occur during the request processing lifecycle and translates them at the end of the request (during the lifecycle's `reportErrors()` phase) to the Struts layer as Struts `ActionError` objects. This means that business validation errors that occur during the processing of the lifecycle neatly appear on the page, wherever we've placed the `<html:errors>` tag(s). However, a failure to include any `<html:errors>` tag in your page will result in errors being reported to the Struts layer, but never displayed. This means that the possible unexpected error will show up only if you explicitly render the Struts errors using the `<html:errors>` tag. When you work with the Data Control Palette in JDeveloper, pages you create will include the `<html:errors>` tag. However, in the event that you create a page entirely in the source code editor, be aware of the need for this tag.

You can optionally complete the following hands-on to explore concepts presented in this lesson.

5.7 Hands-On for Lesson 2

The following hands-on shows how you can easily modify Oracle ADF bindings in your JSP page.

1. In the Application Navigator, expand the **Web Content/WEB-INF/jsp** folder of the **ToyStoreViewController** project and double-click the **showproduct.jsp** file.
2. In the design view of the open file, right-click inside the table cell with the JSTL element for `${Row.Name}` and choose **Table > Split Cell**.
3. In the Split Cells dialog, enter the value `2` for the number of columns to create and click **OK**. You should see a new empty cell to the right of the JSTL `${Row.Name}` table cell.
4. Click the **Source** tab of the visual editor and locate the inserted `<td>&nbsp;</td>` for the new table cell.
5. Replace `&nbsp;` with the JSTL tag `<c:out value="${Row.InStock}"`. Be sure to use the correct case when naming the binding (`InStock`) since binding names are case-sensitive.
6. In the header section of the HTML table element, insert the header for the new column: `<th><font color="white" size="3"><bean:message key="cart.availability"/></font></th>`. The new header should appear after the header defined by the binding `${bindings.ItemsForSale.labels.Name}`.
7. With the `showproduct.jsp` page displayed, open the Structure window and select the **UI Model** tab to view the list of bindings used by this page. (Note that the current 10.1.2 Toy Store project contains extra unused bindings: **Name** and **Listprice**. These two bindings may be safely deleted by choosing **Delete** from their context menus.)
8. Double-click the **ItemsForSale** table binding to display the binding editor.
9. In the binding editor, move the attribute **InStock** from the list of **Available Attributes** to the list of **Display Attributes** and click **OK**.

10. From the Struts page flow diagram, right-click the **home.do** action and choose **Run** to launch the application. Click on any category in the home page. The **Availability** column you added appears in the products in the category page.

The following hands-on shows how you can easily create a new Oracle ADF binding in your JSP page.

1. In the Application Navigator, expand the **Web Content/WEB-INF/jsp** folder of the **ToyStoreViewController** project and double-click the **showproductdetails.jsp** file.
2. In the design view of the open file, right-click inside the table cell with the JSTL element for `${bindings.Name}` and choose **Table > Split Cell**.
3. In the Split Cells dialog, enter the value `2` for the number of columns to create and click **OK**. You should see a new empty cell to the right of the JSTL `${bindings.Name}` table cell.
4. With the `showproductdetails.jsp` page displayed, open the Data Control Palette and expand the **ToyStoreService** data control to display **ProductList** and **ItemsForSale**.
5. Locate the **Productid** attribute node under **ItemsForSale** and drag the attribute node into the new table cell.
6. Return to the **UI Model** tab displayed in the page's Structure window and view the new **Productid** attribute binding.

   Note: As an alternative to using the Data Control Palette, you can also work with the **UI Model** tab to create bindings (using the right-click context menu) and then reference these bindings using JSTL tags in the page source. However, when you work with the Data Control Palette, the IDE performs these steps for you.
7. Display the source view of the `showproductdetails.jsp` page and examine the new table cell definition: `<c:out value="${bindings.Productid}"/>`.
8. From the Struts page flow diagram, right-click the **home.do** action and choose **Run** to launch the application. Click any category in the home page. Select any product link. The product ID you added appears in the products information page.

The next lesson describes available Oracle ADF binding metadata that allow you to customize the behavior of bindings.

6 Lesson Three: Assembling the Shopping Cart

This chapter contains the following sections:

6.1 Introduction

In Lesson 2, we examined the initial pages of the Oracle ADF Toy Store application that permit the user to browse the product catalog. However, we have not yet addressed how the application will assemble the user's shopping cart prior to processing an order. In this lesson, we'll examine how to:

- Extend the Struts page flow to manage the shopping cart and process an order
- Handle updates to the shopping cart when the user removes an item or changes item quantities
- Capture the shopping cart changes in the business model

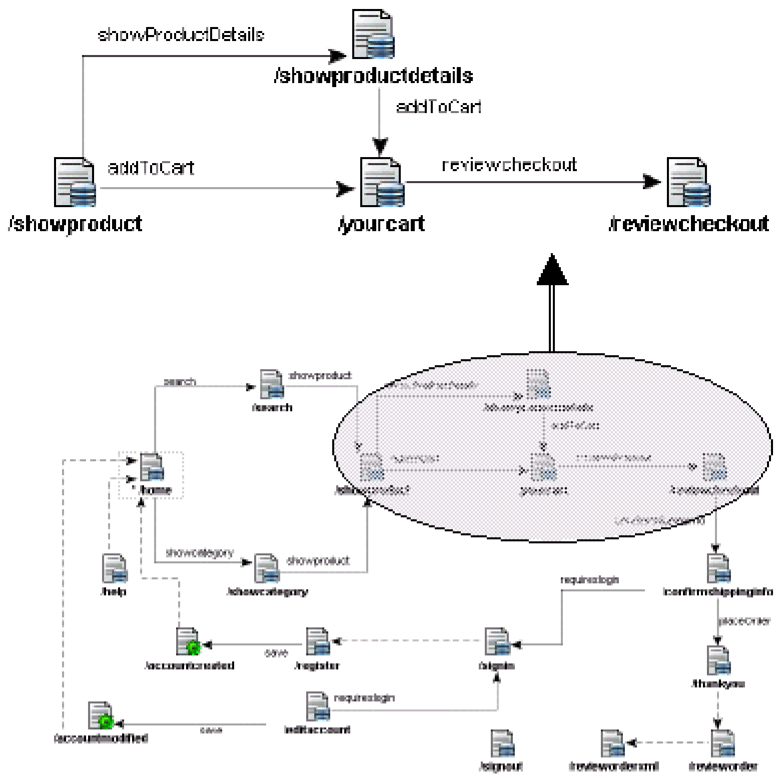
Let's begin by returning to the application workbench, the Struts page flow diagram, to extend our page flow based on these requirements.

6.2 Analyzing the Shopping Cart Page Flow

The page flow proceeds with the action forward `addToCart` from two data pages: `/showproduct` and `/showproductdetails`. The forward destination is the `/yourcart` data page. The `/yourcart` data page will respond to adding and removing products by updating the cart.

Following the `/yourcart` data page, we display the actual order before completing checkout. The page flow now includes a new data page for each task and is shown in Figure 6–1, "Shopping Cart Page Flow".

Figure 6–1 Shopping Cart Page Flow



We continue to use the Oracle ADF data page construct to post back to data actions from mapped JSP pages. The data page's underlying data action ensures that the Oracle ADF model layer is updated before the controller displays the target databound JSP page.

6.3 Managing the State of the Shopping Cart

The goal of managing the contents of the shopping cart is to populate the rows of the cart programmatically as the user adds and removes items from the cart. In the `/yourcart` data page, there is no database query involved in rendering the page. Instead, the shopping cart is implemented as an Oracle ADF view object named `toystore.model.dataaccess.ShoppingCart` in the ToyStoreModel project. This view object has all transient attributes (and no SQL query).

As we'll see in the following sections, the `YourCartAction` action class calls the `ToyStoreService` business service method `adjustQuantitiesInCartAsStringArrays()` to add, change, or remove items from the cart. Since the application relies on the Oracle ADF Business Components stateful mode, the application code does not have to worry about how to store the pending shopping cart data. Examine the Tuning page of the View Object Editor for the `ShoppingCart` view object. There we indicate declaratively that all transient attributes of this view object will be passivated by the framework's state management mechanism. Just checking the checkbox there is the only work required to leverage this feature.

On each subsequent request, our actions can access the `ShoppingCart` object and programmatically adjust the contents of its default rowset.

6.4 Handling the Product Pages' Add-to-Cart Event

In the previous pages, we saw how to pass the `id` of the product on the request object using an HTML link and the `<c:url>` tag to specify a named event. The `addToCart` action is another such event, and it is triggered with this syntax from the `showproductdetails.jsp` and the `showproduct.jsp` pages:

```
<a href="<c:url
            value='showproductdetails.do?event=addToCart&id=${bindings.ItemId}'/>"
             ><img src="<bean:message key='images.buttons.addtocart'/>" border="0"
                    alt="<bean:message key="cart.addItem"/>"> </a>
```

While the previous pages' actions handled no events, we want the product pages' actions to update the quantity attribute of the cart business object. Whenever the user clicks the **Add to Cart** image displayed by the above link, the controller executes the product page's corresponding data action and its `onAddToCart` event handler:

```
public void onAddToCart(DataActionContext ctx) {
   String[] id = new String[] { ctx.getHttpServletRequest().getParameter("id") };
   String[] qty = new String[] { "1" };
   getToyStoreService(ctx).adjustQuantitiesInCartAsStringArrays(id, qty);
}
```

Again, we see the use of a service method to encapsulate business logic, where the `adjustQuantitiesInCartAsStringArrays()` method is implemented. We'll use this same method to handle removing items and updating quantities in the displayed shopping cart page, as described next.

6.5 Handling the Shopping Cart Page's Add/Remove Events

The Struts controller displays the shopping cart page (`yourcart.do`) anytime the user clicks the **Add to Cart** link for a specific product. The shopping cart page comprises the same product list table with the `<c:forEach>` loop used in the product by category page. However, this time we render the table as the body of an HTML `<form>` element in order to accept the user's quantities on individual items in the cart:

```
<form action="<c:url value='yourcart.do'/>" method="post">
```

The form posts back three pieces of information on the response object, where it can be retrieved by the `/yourcart` action for processing:

- The `event_updateCart` named event, specified by the Update Totals input
- The `Itemid` of the current product being iterated on by the `<c:forEach>` loop, specified by a hidden input element
- The `Quantity` to assign to the current product of the `<c:forEach>` loop, specified by a text edit input element

To trigger the quantities update, the user presses the **Update Totals** button. The event is submitted on the `name` attribute of the input element:

```
<input type="image" src="<bean:message key='images.buttons.updatecart'/>"
       name="event_updateCart">
```

Open the file `YourCartAction.java`, which implements the data action (in the page flow diagram, right-click **/yourcart** and choose **Go to Code**), and find the `onUpdateCart()` event handler. First, the event handler obtains the product `id` of the current row and the product quantity entered by the user:

```
String[] id = ctx.getHttpServletRequest().getParameterValues("id");
```

```
String[] qty = ctx.getHttpServletRequest().getParameterValues("qty");
```

Next, the update cart event handler calls the `adjustQuantitiesInCartAsStringArrays()` service method to modify the cart quantities.

A similar event handler (`onRemoveItem`) exists to reset the cart quantity to zero when the user clicks the **Remove** button. In this case, the event is submitted with a URL and no form input is submitted:

```
<a href="<c:url value='yourcart.do?event=removeItem&id=&{Row.Itemid}'/>">
   <img src="<bean:message key='images.buttons.removefromcart'/>" border="0" ></a>
```

After the data action executes the `onUpdateCart` or `onRemoveItem` event handler, the `invokeCustomMethod()` method is executed to prepare the cart total for display by `yourcart.jsp`. Unlike the shopping cart's `ExtendedTotal` attribute (which is calculated based on item quantities), the cart total is not a business service attribute to be accessed by the Oracle ADF shopping cart table binding. In order to make the result of the `getCartTotal()` method available to the JSTL tags in the shopping cart page, the `invokeCustomMethod()` adds the cart total as a request attribute:

```
protected void invokeCustomMethod(DataActionContext ctx) {
    Double cartTotal = getToyStoreService(ctx).getCartTotal();
    ctx.getHttpServletRequest().setAttribute("CartTotal", cartTotal);
}
```

The action mapping for the `/yourcart` action completes the postback pattern by invoking the `yourcart.jsp` page to render the form with the prepared data. You can confirm the target page by examining the `struts-config.xml` file or by mousing over the `/yourcart` action in the page flow diagram.

Note that the `yourcart.jsp` page does not use the Struts `<html:form>` and `<html:input>` elements to render the quantity input fields. In this example of form input, a programmatic approach was taken that simplifies handling the multiple form inputs provided by the user. Currently, Oracle ADF data binding provides no built-in support for easily handling multi-row input forms. By creating a String array with the product IDs and quantities, the `adjustQuantitiesInCartAsStringArray()` business service method (see `ToyStoreServiceImpl.java` in the `toystore.model.services` package) encapsulates the task to greatly simplify the `onUpdateCart` event handler. In Lesson 5, we'll see the use of a Struts form to display and modify the customer's account.

6.6 Handling the Shopping Cart Page's Review-Checkout Event

Once the user is satisfied with the cart, they can initiate their purchase by triggering the `reviewcheckout` forward from the shopping cart page. The **Proceed to Checkout** button sends the event `reviewcheckout`:

```
<a href="<c:url value='yourcart.do?event=reviewcheckout'/>">
    <img src="<bean:message key='images.buttons.checkout'/>"
         alt="Proceed To Checkout" border="0"> </a>
```

Notice that the action forward `reviewcheckout` (shown in the page flow diagram) and the event share the same name. By default, after the Oracle ADF data action executes, the next phase of the Oracle ADF lifecycle will return the name of the action forward to use. If this check returns null, meaning that the developer has not previously programmatically set the action forward, then as a useful fallback behavior, the Oracle ADF lifecycle will try to find a forward with the same name as the current event being handled. If such a forward with a matching name exists, the default `findForward()` implementation will set that action forward to be used.

This explains why, without writing any custom controller code, the user can click the **Proceed to Checkout** button and see the `reviewcheckout.jsp` page. The HTML form submits the `reviewcheckout` operation and the Oracle ADF lifecycle uses the matching `reviewcheckout` forward to perform declarative navigation.

6.7 Working with Operations in Forms

The Oracle ADF named events let you handle multiple operations initiated from HTML form inputs by writing event handlers in the Oracle ADF data action. The result is a more declarative implementation of the Struts `ForwardAction` class. The naming conventions for operations (named events) differ from those Struts uses for the dispatch parameter in the `DispatchAction` or `LookupDispatchAction` objects. In a Struts-only application, the dispatch parameter (used to identify the operation corresponding to an HTML form submit element) can have any desired name. However, in ADF, the operation submitted by the input element's name attribute is usually specified as `event_`*`[operationName]`*. The following discussion provides examples.

You can use the HTML image input element and set the `name` attribute to specify the operation like this:

```
<input type="image" src="<bean:message key='resource key'/>"
       name="event_[operationName]">
```

Or, you can use the HTML submit button similarly:

```
<input type="submit" name="event_[operationName]" value="message resource key"/>
```

Or, you can use a URL, with the event parameter and operation name:

```
<a href="<c:url value=actionName.do?event=[operationName]/>">link text or image
source</a>
```

Finally, in the data action, the method to handle a given operation is named like this:

```
public void on[operationName] (DataActionContext actionContext)
```

As long as the *`[operationName]`* parts match, the event and a handler will be bound together at runtime. It is important to remember that the operation name must match the action forward name and that it is case-sensitive. Additionally, the event method handler must also match. For example:

```
event="foo"

name="event_foo"
```

will match an action forward named `foo` but not `Foo` or `FOO`.

The method to fire must be named `onFoo()`, with the convention that the term after "on" is always initial capped regardless of the event name (thus `onfoo()` is not valid and the events `foo` and Foo both fire the same method onFoo() in the data action).

> **Best Practice Tip:** Since the event handler method does not distinguish between `foo` and `Foo` operation names, it is best to avoid names that differ only by letter capping. Otherwise, more than one operation name would fire a single event handler method.

Best Practice Tip: In the case of submit buttons, the value attribute can alternatively be used to name the operation, but this attribute could change depending on the locale and translation string. Oracle ADF data action provides a simple approach: instead of using the value attribute of the HTML element to represent the operation name to submit, alternatively use the "`event_`" prefix in the name of the button.

That is, rather than writing:

```
<input type="submit"
       name="event"
       value="<bean:message key='button.add.one'/>"/>
```

to submit an operation whose name is given in the value of the button (the label "`Add One`"), write instead:

```
<input type="submit"
       name="event_Increment"
       value="<bean:message key='button.add.one'/>"/>
```

changing the `name="event"` to `name="event_Increment"`. This way, regardless of the value of the button (or whether that value contains spaces in the name), you'll have a reliable way to submit the event without running into issues.

Note: To learn more about how Oracle ADF event handling integrates with Struts, see the `Oracle ADF Data Binding Primer and ADF/Struts Overview` whitepaper on OTN.

You can optionally complete the following hands-on to explore concepts presented in this lesson.

6.8 Hands-On for Lesson 3

The following hands-on demonstrates a practical example of the Oracle ADF framework's state management and failover support for transient view objects in ADF Business Components. You can run the Oracle ADF Toy Store application inside JDeveloper's embedded OC4J container and try the following, but you must first enable failover for the ADF Business Components:

To enable failover mode for the Oracle ADF Toy Store application, select **ToyStoreService** in the Application Navigator and choose **Configurations** from the context menu. In the dialog, select the **ToyStoreServiceLocal** configuration and click the **(Edit)** button. In the Pooling and Scalability page, select the **Failover Transaction State Upon Managed Release** property.

Now that you have enabled failover, return to the Toy Store application:

1. Add several items to your shopping cart.
2. Without closing your browser window, terminate the OC4J application server to simulate a hardware failure on your application server machine.

   To do this, choose **View > Run Manager** to display the Run Manager. Find the **Embedded OC4J Server** process in the list, and select it. Finally, choose **Terminate** from the context menu.
3. Rerun the Oracle ADF Toy Store application.

After you restart the application server — in this example, we've restarted the embedded OC4J application server in JDeveloper — the browser window you left open in step 2 above will be able to continue where it left off, with all shopping cart items intact.

If you're still curious, try disabling failover support for the `ShoppingCart` component in JDeveloper:

1. In the Application Navigator, expand the **toystore.model.dataaccess** package of the **ToyStoreModel** project and double-click the **ShoppingCart** view object.
2. In the View Object Editor, select the Tuning page.
3. In the Tuning page, deselect **Including All Transient Values**.
4. Terminate the embedded OC4J process and rerun the application.
5. Return to the shopping cart page and receive the exception `javax.servlet.jsp.JspException: Missing message for key "cart.instock."`.

The page fails to render the message because `<c:set>` fails to evaluate the value expression before rendering `inStockMsgKey` from the resource bundle:

```
<c:set var="inStockMsgKey"
       value="cart.instock.${Row.InStock}"/>
<bean:message name="inStockMsgKey"/>
```

Failover support works because the Oracle ADF framework offers automatic database-backed state management for pending data in your application when you use ADF Business Components. In the Oracle ADF Toy Store application, the pending shopping cart information is not stored in the HTTP session state as it is in most applications. Instead, with a declarative checkbox on the `ShoppingCart` component at design time, we indicate that we'd like this component's pending data to be managed for us. And the framework takes care of the rest.

7 Lesson Four: Processing the Shopping Cart Order

This chapter contains the following sections:

7.1 Introduction

With the ability to add products to the cart, as described in Lesson 3, we're ready to extend our page flow to take the customer's shipping information and provide confirmation of their order. To accomplish this, we will examine the structure of two more JSP pages:

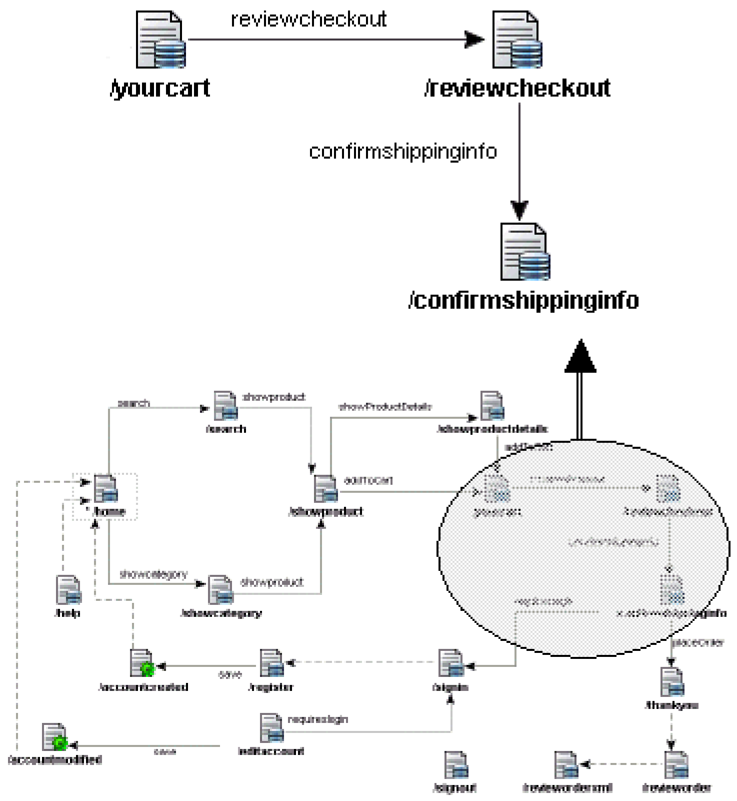
- `reviewcheckout.jsp`, to display the customer's order for confirmation
- `confirmshippinginfo.jsp`, to let the customer enter the shipping information and payment method

In this lesson, we'll examine the layout of these two new JSP pages.

7.2 Analyzing the Proceed to Checkout Flow

The page flow proceeds with the action forward `reviewcheckout` from the `/yourcart` data page. The forward destination is the `/reviewcheckout` data page. The `/reviewcheckout` data page's only function is to let customers preview their final order. The only event possible from this page is `confirmshippinginfo`, which maps to the next forward and causes the `confirmshippinginfo.jsp` page to be displayed. The page flow now includes a new data page for each task and is shown in Figure 7–1, "Proceed to Checkout Page Flow".

Figure 7–1 Proceed to Checkout Page Flow



We'll reserve the action mapping discussion for the `confirmshippinginfo.do` action until the next lesson, when we also describe the target pages in the flow diagram.

Using the page flow modeler, in actual practice, proceeds roughly as follows:

1. Identify the pages and actions of your flow.
2. Identify the events (and action forwards) of your flow.
3. Using the Component Palette, drop the page source and target into the diagram area of the Struts Page Flow Modeler.
4. Using the Component Palette, drop an action forward element (or page link) to connect the source and target elements.

7.3 Laying Out the Review Checkout Page

At runtime, the `reviewcheckout` forward initiated from the `yourcart.jsp` page executes the mapping for the `/reviewcheckout` data page. In this case, the data action corresponding to this page performs no model initialization. Unlike the previous pages, where model initialization was necessary to prepare data for the page to display, the `reviewcheckout.jsp` page will only access existing model objects. The purpose of this page is merely to pull data from the existing `ShoppingCart` object and present it in a concise, readable table.

Specifically, the `reviewcheckout.jsp` page uses the `<c:forEach>` tag to iterate over the Oracle ADF table binding bound to the `ShoppingCart` object through the `ShoppingCartIterator` binding. Data for the table rows is displayed using a variety of tags that take input from the table binding's current `Row`:

When the attribute to be displayed for the current row does not use a formatter to render the data, we use the standard `<c:out>` tag:

```
<td><c:out value="${Row.Itemid}" /></td>
<td><c:out value="${Row.Name}" /></td>
...
<td align="center"><c:out value="${Row.Quantity}" /></td>
```

When the attribute to be displayed relies on language-sensitive format masks that were defined in the business components, we use the Oracle ADF tag `<adf:render>`:

```
<td align="right"><adf:render model="Row.Listprice" /></td>
<td align="right"><adf:render model="Row.ExtendedTotal" /></td>
```

Finally, when we want to render a translatable string based on single-character flag values, such as "Y" or "N" for the `InStock` attribute, we use the Struts tag `<bean:message>`:

```
<td>
   <c:set var="inStockMsgKey" value="cart.instock.${Row.InStock}"/>
   <bean:message name="inStockMsgKey"/>
</td>
```

The `ShoppingCart` object contains a simple transient field named `InStock`, which takes the value either `Y` (yes, in stock) or `N` (no, not in stock) to indicate whether the item is available. When the `reviewcheckout.jsp` page displays the `InStock` information, rather than showing the raw Y or N value, we use the Y or N as part of the string key name for a translatable string in the Struts message resource file. The above code first uses the JSTL `<c:set>` tag to set a local page variable named `inStockMsgKey` to the value of `cart.instock` concatenated to the value of the `InStock` field in the current `Row` of the `<c:forEach>` loop, and then it uses `<bean:message>` to display the translated string based on either the `cart.instock.Y` or `cart.instock.N` message key value in that `inStockMsgKey` object. This way, the user can see a meaningful indicator of availability in the language specified by the browser preference setting. For instance, the Y can display as `In Stock` in English or `In Magazzino` in Italian.

The last row of the table displays a double-valued attribute, `CartTotal`, using the `<bean:write format="$0.00">` tag:

```
<bean:write format="$0.00" name="CartTotal"/>
```

Note: The decimal formats `$0.00` and `#,##0.00` are equivalent, and either may be used to represent the currency value.

The value of `CartTotal` is made available to the page when `invokeCustomMethod()` is executed on the data action:

```
protected void invokeCustomMethod(DataActionContext ctx) {
     Double cartTotal = getToyStoreService(ctx).getCartTotal();
     ctx.getHttpServletRequest().setAttribute("CartTotal", cartTotal);
}
```

This use of the `<bean:write>` tag with format attribute illustrates an alternative to the `<adf:render>` approach described for the `Listprice` and `ExtendedTotal` attribute values. With `<adf:render>`, format masks are provided as hints on the Oracle ADF business object's attributes in the model layer. With `<bean:write>`, the format masks are hard-coded into the pages (or with the `<bean:write> formatKey` attribute, they can be specified as translatable message keys).

Once the customer is satisfied with the order, they check out by triggering the `confirmshippinginfo` forward. The **Continue** button sends the event `confirmshippinginfo`:

```
<a href="<c:url value='reviewcheckout.do?event=confirmshippinginfo'/>"
             ><img src="<bean:message key='images.buttons.continue'/>"
                    alt="Continue" border="0"></a>
```

As we saw in Lesson 3, again the action forward `confirmshippinginfo` (shown in the page flow diagram) and the event share the same name. Although the data action doesn't explicitly set the action forward, the Oracle ADF lifecycle will automatically seek a matching forward and find `confirmshippinginfo` to perform declarative navigation.

7.4 Laying Out the Confirm Shipping Information Page

The main design feature of the `confirmshippinginfo.jsp` page is the use of a Struts form (`<html:form>` tag) to populate the form and accept user input. The fields of the form are also Struts HTML elements, which access the properties of the form bean:

- `<html:text>`, where the `property` attribute is the name of the Oracle ADF attribute binding that accesses the desired business object
- `<html:select>`, where the `property` attribute is the name of the Oracle ADF list binding that accesses the desired business object and `<html:optionsCollection>` populates a poplist

The `<html:select>` and `<html:optionsCollection>` tags work with Oracle ADF list bindings to populate the poplist in the page. The `<html:select>` tag is bound to a property of the form bean, which shares the same name as the list binding object (`Cardtype` and `ExprYear`). The `<html:optionsCollection>` tags get their data from the nested, list-valued `displayData` property of the Oracle ADF list binding. The beans in these display data collections each have a `prompt` and an `index` property, so we indicate to use those as the label and value (respectively) for each option in the list. In this sample, we show the selection list for the charge card type (corresponding to the `Cardtype` bean and binding object):

```
<html:select property="Cardtype" >
       <html:optionsCollection label="prompt" value="index"
                              property="Cardtype.displayData" />
</html:select>
```

Note: For bandwidth optimization, the Oracle ADF binding layer expects the nonvisible values of the Oracle ADF list binding to be the zero-based `index` number in their `displayData` collection. The Oracle ADF list binding handles translating the underlying list of values (like `IT` for a country code) into index positions (like `86`) on both read and write of the binding value.

In Oracle ADF, the HTML form is tied to the associated action, which is tied to the data form bean. At runtime, the HTML form relies on the data form bean to resolve the properties corresponding to the Oracle ADF binding objects and thereby to display available form attribute values:

1. First, the data form bean asks the Oracle ADF binding container (`BindingContainerActionForm`) whether it has a binding with a name matching the form attribute.
2. Next the Oracle ADF binding container returns the binding if it exists; and, finally, the data form bean populates the HTML form attributes with that binding's value.

Similarly, when the user clicks the **Continue** button to submit the form with its data, thereby submitting the `placeOrder` event, the Oracle ADF binding container collects the form attribute values that Struts has set on it; then, during the `processUpdateModel()` phase of the lifecycle, the binding container uses those values to update the Oracle ADF binding objects with matching names.

However, if the user submits the form and validation errors in the model layer are thrown, when this page is rendered again, the `<html:errors>` tag will ensure that errors related to the attributes will show up next to the fields.

Note: The Oracle ADF business object that represents a user account in the model layer is declaratively enforcing mandatory attributes, reusing a custom business rule to validate the country and state combination, using a built-in validation rule to enforce uniqueness of the primary key attribute, and validating the correct formatting of email addresses using a custom `Email` datatype. All of the custom error messages are localized to the current browser user's locale (based on language + territory). None of this behavior requires developer-written code to coordinate.

Best Practice Tip: By default, the Oracle ADF data actions bundle all exceptions that occur during the request processing lifecycle and translate them at the end of the request (during the lifecycle's `reportErrors()` phase) to the Struts layer as Struts `ActionError` objects. This means that business validation errors that occur during the processing of the lifecycle neatly appear on the page, wherever you've placed the `<html:errors>` tags. However, a failure to include any `<html:errors>` tag in your page will result in the errors being reported to the Struts layer, but never displayed. This means that even an unexpected error will show up only if you explicitly render the Struts errors using the `<html:errors>` tag. When you work with the JDeveloper Data Control Palette, you will see `<html:errors>` tags in your page, but if you develop your pages in a more manual way, be aware that you must add these tags yourself.

The form includes the standard hidden field that the Oracle ADF controller layer uses to detect whether the user has tried to submit the same form multiple times in rapid succession:

```
<input type="hidden" name="<c:out value='${bindings.statetokenid}'/>
       "value="<c:out value='${bindings.statetoken}'/>"/>
```

The token prevents the data action from processing the same request multiple times should the user page back and forward.

You can optionally complete the following hands-on to explore concepts presented in this lesson.

7.5 Hands-On for Lesson 4

The following hands-on shows how Oracle ADF Business Components supports the use of validation domains to specify an attribute datatype to perform custom validation on the databound attribute.

1. In the Application Navigator, locate **toystore.model.datatypes**, right-click, and choose **New Domain**.
2. In the Create Domain wizard, click **Next** and enter the name `Phone` for the domain name. Leave all other options unchanged and click **Finish**.
3. In the Application Navigator, right-click the new domain **PhoneNumber** and choose **Go to Domain Class** to open the `Phone.java` template you will modify.
4. In the open source for `Phone.java`, replace the stub `validate()` method with the following validation code:

```
protected void validate() {
    if (!isEightCharacters()) {
      throw new DataCreationException(ErrorMessages.class,
        ErrorMessages.INVALID_PHONENUMBER, null, null);
    }
  }
  private boolean isEightCharacters() {
    if (mData != null) {
      if (mData.length() != 8) {
        return false;
      } else {
        return true;
      }
    }
    return false;
 }
```

5. Add the following import statement to the list of imports:

```
import oracle.jbo.domain.DataCreationException;
```

6. Make no other changes to the file and choose **File > Save**.
7. In the Application Navigator, expand **toystore.model.datatypes.common** and double-click **ErrorMessages.java** to add the error message for `INVALID_PHONENUMBER` specified in `Phone.java`.
8. In the open source for `ErrorMessages.java`, add this declaration to the list of constants:

```
public static final String INVALID_PHONENUMBER = "20005";
```

9. To the list of error messages add:

```
{ INVALID_PHONENUMBER, "Phone number must be eight characters including dash."
},
```

10. In the Application Navigator, expand **toystore.model.dataaccess**, right-click **Accounts**, and choose **Edit Accounts**.
11. In the View Object Editor, select **Attributes** to display the list of available and selected attributes. Select **Phone** in the **Selected Attributes** list and click the **Remove** arrow button to return it to the **Available Attributes** list. This step is necessary to remove the dependency of the view object on this attribute before we can apply the new domain type. Click **OK** to save the changes.

12. In the Application Navigator, expand **toystore.model.businessobjects**, right-click **Account**, and choose **Edit Account**.
13. In the Entity Object Editor, expand **Attributes** and select **Phone** from the list.
14. In the **Entity Attribute** tab, display the **Type** dropdown list and choose **toystore.model.datatypes.common.Phone**. If the new datatype is not displayed, you can also select **Import Domain** from the list to locate the domain. Click **OK** to save the changes and exit the editor.
15. In the Application Navigator, expand **toystore.model.dataaccess**, right-click **Accounts** and choose **Edit Accounts**.
16. In the View Object Editor, select **Attributes** to display the list of available and selected attributes. Select **Phone** in the **Available Attributes** list and click the **Add** arrow button to return it to the **Selected Attributes** list. Click **OK** to save the changes and exit the editor.
17. Right-click the **home.do** action and choose **Run** to launch the application. Click any category in the home page. Select any product link and add it to your cart. Proceed to checkout. When asked to sign in, click the **Register as a New User** link instead. Complete the form and enter an intentionally short phone number. Then click **Submit**. The data action redisplays the page with the validation error you created for the phone number field.

Validation domains add business logic to every attribute that uses a validation domain as its type. Domain validation occurs when an object of that domain type is created. The data object can then be passed between the tiers without the need for reconstruction or revalidation. To implement domain-level validation, you create a new domain type and add code to the `validate()` method. Once the domain has been created, you can assign it as the type of an attribute.

8 Lesson Five: Requiring the User to Sign Into an Account

This chapter contains the following sections:

8.1 Introduction

The pages discussed in Lesson 4 demonstrate how to process the customer order. In this lesson, we examine how to require the customer to sign in to an existing account, and, in the case of a new customer, how to create an account. To accomplish this, we will examine the structure of these JSP pages:

- `registernewuser.jsp`, to take a new customer's account information
- `signin.jsp`, to accept an existing customer's user ID and password
- `accountcreated.jsp`, to confirm the new customer's account creation

In this lesson, we'll examine the layout of these new JSP pages and describe how the application returns the customer to the previous page after sig- in is verified.

8.2 Analyzing the Sign-In Page Flow

The page flow proceeds with the action forward `requireslogin` from the `/confirmshippinginfo` data page. The forward destination is the `/signin` data page. In this application, we have choosen to require sign-in before taking the customer's shipping information. Although this task could have been performed earlier, we prefer not to interfere with the customer's ability to browse the catalog until they proceed to checkout. To ensure that the customer does sign in, the `initializeModelForPage()` method in the `confirmshippinginfo` data action performs this test:

```
protected void initializeModelForPage(DataActionContext ctx) {
   HttpServletRequest request = ctx.getHttpServletRequest();
```

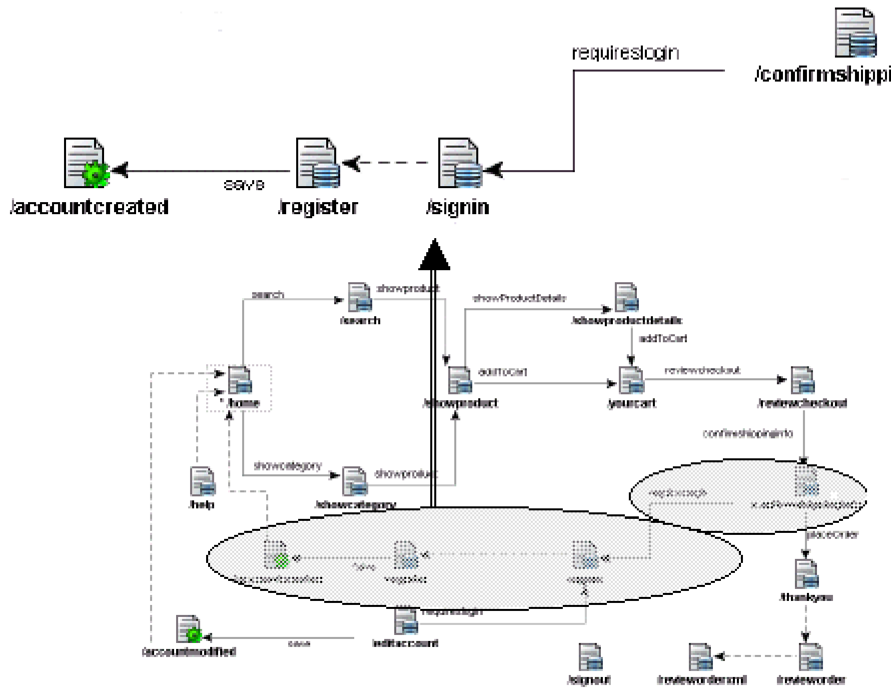
```
    if (!AppUserInfo.isSignedOn(request)) {
        ctx.setActionForward("requireslogin");
    } else {
        getToyStoreService(ctx).createNewOrder(AppUserInfo.signedInUser(request));
      }
}
```

If the user has signed in already, the order is created by the service method `createNewOrder()` and the application displays the `confirmshippinginfo.jsp` page, as described in Lesson 4. However, assuming that the customer has not signed in, the Struts method `setActionForward()` is invoked with the forward `requireslogin` to initiate the action mapping and display the `signin.jsp` page.

After programmatically invoking the action forward and displaying the `signin.jsp` page, the user must be able either to sign in or to register as a new customer for the first time. The page flow diagram represents the transition from `signin.jsp` to `registernewuser.jsp` as a dashed line to distinguish an ordinary page link from an action forward (represented by a solid line). The difference is that the page link does not map to an associated data action and therefore a link cannot cause the application to attach the binding container to the target page. As we will see, the register new user page is an empty form with no data displayed.

The last task of the sign-in page flow is to provide confirmation to the user once they have registered a new account. The account confirmation task is represented in the page flow diagram by the `save` action forward with the target **/accountcreated** page forward icon. The final diagram with the sign-in page flow is shown in Figure 8–1, "Sign-In Page Flow".

Figure 8–1 Sign-In Page Flow



The page forward has not yet been used in the Struts page flow of this application, but it is conceptually similar to a page link: in either case the target page requires no data binding. In the case of a page forward, the icon is a representation of a Struts action that always forwards to a specified destination web page.

Best Practice Tip: Although the Struts page flow diagram allows you to insert JSP page icons into the diagram, in most cases you will use the Page Forward icon instead of a JSP icon to represent the target web page. This standard practice ensures that all web pages are represented in the Struts configuration file and therefore permits the Struts controller to handle page navigation.

We'll reserve the action mapping discussion for the `confirmshippinginfo.do` action until the next lesson, when we also describe the target pages in the flow diagram.

8.3 Laying Out the Sign-In Page

The main design feature of the `signin.jsp` page is the use of a Struts form (`<html:form>` tag) to accept user input. The fields of the form are also Struts HTML elements that access the properties of the form bean:

- `<html:text>`, where the `property` attribute is the name of the Oracle ADF attribute binding that accesses the desired business object
- `<html:password>`, where the `property` attribute is the name of the Oracle ADF attribute binding that accesses the desired business object

When the user clicks to submit the form, with these additional attributes set, at runtime the Oracle ADF binding container populates the bindings with the values of the username and password HTML form fields. Those values are accessible by the `onVerifySignin()` event-handler method of the `SignInAction` class, which will handle the sign-in form's postback when the user submits the form.

The `onVerifySignin()` method in the data action first verifies that neither the `username` nor `password` properties is blank, and then calls the `validSignon()` method on the `ToyStoreService` business service interface to verify whether the username/password combination represents a valid web store user. If any validation check fails, `onVerifySignin()` adds a Struts `ActionError` object to the `ActionErrors` collection so that the view layer can present appropriate error messages to the user and so that the user is returned to the sign-in page to try again. If the sign-in validation check *succeeds*, then the `onVerifySignin()` method calls a helper method (`signIn()` method of the `AppUserInfo` class) to flag the current user as signed in, and returns the appropriate page to forward the request to.

Since several different actions in the application can require the user to log in, the `setForwardAction()` method in this data action uses the `target` parameter to return the correct "next" page in the flow, based on which action required the user to log in.

Best Practice Tip: With the addition of a secondary resource message file for global errors identified by the key `GlobalErrors`, you can ensure that error strings like those represented by the `INVALIDLOGIN` constant are translated into user-readable messages. You make Struts aware of the names of your message resource files in `struts-config.xml`, where the `<message-resources>` element with the key attribute defines the location of the secondary message resource:

```
<message-resources key="GlobalErrors"
                   parameter="toystore.view.GlobalErrors"/>
```

8.4 Laying Out the Register New User Page

The `registernewuser.jsp` page (used by the `/register` data page) renders the data entry form that allows users to register on the site for the first time. Because the results produced in the browser of this page are nearly identical to the `editexistingaccount.jsp` page, we decided to render the entire form by the single `<jsp:include page="formControl.jsp">` tag. This tag works like a reusable component, including the contents of the `formControl.jsp` page. The nested `<jsp:param>` tags pass three parameters to the reusable component page:

- `dataPage` — the name of the current data page
- `saveButtonLabelKey` — the message bundle key to the label to be displayed on the **Save** button
- `saveButtonEvent` — the name of the event to be associated with the clicking of the **Save** button.

So the actual work being done lies in the `formControl.jsp` component page. The page builds a data entry form with one databound control for each control value binding in the current binding container.

Note: The following discussion represents a generic, metadata-driven way of rendering the binding data in contrast to the more traditional approach of specifying each binding in the page source. In contrast to this generic technique explained below, the Toy Store application also includes the register new user page to render a data entry form in the traditional way. Both forms render the same set of controls for account data, so you can compare the two approaches and pick the one that will suit your application needs best. See Lesson 6 for a discussion of the register new user page.

8.5 Laying Out the Generic Form Control Page

The form control page begins with some examples of using the `<c:choose>`, `<c:if>`, and `<c:set>` tags. The following excerpt uses these tags to conditionally set up the values of local page variables named `eventName`, `buttonLabel`, and `buttonLabelKey`, based on whether and which of the expected input parameters were provided. We'll use these variables later in the page as part of constructing the **Save** button at the bottom of the generated form.

```
<c:choose>
   <c:when test="${not empty param.saveButtonEvent}">
     <c:set var="eventName" value="${param.saveButtonEvent}"/>
   </c:when>
   <c:otherwise>
     <c:set var="eventName" value="Commit"/>
   </c:otherwise>
</c:choose>
<c:if test="${not empty param.saveButtonLabel}">
   <c:set var="buttonLabel" value="${param.saveButtonLabel}"/>
</c:if>
<c:if test="${not empty param.saveButtonLabelKey}">
   <c:set var="buttonLabelKey" value="${param.saveButtonLabelKey}"/>
</c:if>
```

The `formControl.jsp` page goes on to use the `<html:errors>` tag as part of a "global errors" section of the input form, where any errors that are not attribute-specific will show up:

```
<center>
   <table border="0">
     <tr>
       <td><html:errors bundle="GlobalErrors"
                      property="<%= ActionErrors.GLOBAL_ERROR %>"/></td>
     </tr>
   </table>
</center>
```

Next, the form uses the value of the `dataPage` parameter passed in by the `<jsp:include>` tag as part of opening the `<html:form>` tag. Notice that since we cannot use EL expressions directly in the `<html:form>` tag's `action` attribute, we first use `<c:set>` to set a local page variable named `name` with the EL-expression value we want, and then we use a JSP scriptlet to pass the value of this `name` variable to the `action` attribute:

```
<c:set var="name" value="${param.dataPage}.do"/>
<html:form action='<%= pageContext.getAttribute("name")%>'>
     <!-- etc. -->
</html:form>
```

The form includes the standard hidden field that the Oracle ADF controller layer uses to detect whether the user has tried to submit the same form multiple times in rapid succession:

```
<input type="hidden" name="<c:out value='${bindings.statetokenid}'/>"
            value="<c:out value='${bindings.statetoken}'/>"/>
```

Next we begin the loop that will create an HTML form field for each control value binding in the binding container. Inside the `<table>` tag, we have the following `<c:forEach>` iteration:

```
<c:forEach var="curBinding" items="${bindings.ctrlBindingList}">
   <% JUControlBinding cb =
        (JUControlBinding)pageContext.getAttribute("curBinding");
      if (cb instanceof JUCtrlValueBinding &&
        !(cb instanceof JUCtrlRangeBinding) &&
        !(cb instanceof JUCtrlHierNodeBinding)) { %>
    <!-- Build control for current control value binding in here -->
  <% } %>
</c:forEach>
```

The `<c:forEach>` loop iterates over the list of control value bindings from the binding container. Since this list might include control *action* bindings, we need to skip over these when rendering the input controls. Since we want to keep things simple, we'll also skip over `RangeBindings` and `TreeBindings` too. The EL expression language doesn't have a built-in `instanceof` operator, so we're using a JSP scriptlet to insert a regular Java-language `if` statement to perform the combination of `instanceof` checks.

Note: We could have decided to generically render a set of buttons for any of the action bindings found in the binding container, which would be of type `JUCtrlActionBinding` in the `oracle.jbo.uicli.binding package`; instead, the application just renders a single **Save** button on the form.

Since we specified the `var="curBinding"` attribute on the `<c:forEach>` tag, inside the loop we can refer to this `curBinding` loop variable to access the current control value binding as part of our generic form input control generation. In the following excerpt, notice how we're making use of the binding properties in our EL expressions like `tooltip`, `mandatory`, and `label` to access this metadata from the current control binding.

```
<c:forEach var="curBinding" items="${bindings.ctrlBindingList}">
   <% JUControlBinding cb =
        (JUControlBinding)pageContext.getAttribute("curBinding");
      if (cb instanceof JUCtrlValueBinding &&
        !(cb instanceof JUCtrlRangeBinding) &&
        !(cb instanceof JUCtrlHierNodeBinding)) { %>
    <tr>
      <th align="right" title="<c:out value='${curBinding.tooltip}'/>">
        <c:if test="${curBinding.mandatory}">*&nbsp;</c:if>
        <c:out value="${curBinding.label}"/>
      </th>
      <td>
         <c:set var="name" value="bindings.${curBinding.name}"/>
         <adf:inputrender model='<%= pageContext.getAttribute("name")%>'/>
      </td>
      <c:set var="name" value="${curBinding.name}"/>
      <td>
        <html:errors property='<%= pageContext.getAttribute("name") %>'/>
      </td>
    </tr>
  <% } %>
</c:forEach>
```

To actually render the HTML form control, we use the `<adf:inputrender>` tag, which is set up to render an appropriate tag based on the datatype of the current binding's attribute value (the use of Business Components metadata to invoke a custom renderer will be discussed in Lesson 6). We repeat our trick of using `<c:set>` to set a local page variable named `name` to the concatenation of the string "`bindings.`" with the name of the current binding, which is what the `<adf:inputrender>` tag expects as the value of its model attribute. We use the `<html:errors>` tag to show any attribute-level validation errors that might occur next to the control to which they are relevant. We again use the `<c:set>` trick to make the value of the `<html:errors>` tag's `property` attribute match the name of the current binding.

Finally, as the following excerpt shows, we use a `<c:choose>` tag to put the appropriately labeled **Save** button at the bottom of the form. Based on whether the user specified a button label or a button label key, we either use the literal label string or employ the `<bean:message>` tag to look up the label key for us. We're using our local page variable `eventName` that we set up at the top of the page to fill in the right name for the button to generate that event when the user clicks it.

```
<c:choose>
   <c:when test="${not empty buttonLabel}">
     <input name="event_<c:out value="${eventName}"/>" type="submit"
```

```
            value='<c:out value="${buttonLabel}"/>'/>
   </c:when>
   <c:when test="${not empty buttonLabelKey}">
     <input name="event_<c:out value="${eventName}"/>" type="submit"
            value='<bean:message name="buttonLabelKey"/>'>
   </c:when>
   <c:otherwise>
      <input name="event_<c:out value="${eventName}"/>" type="submit"
            value='Submit'/>
   </c:otherwise>
</c:choose>
```

8.6 Laying Out the Account Created Page

As previously explained, the Struts page flow uses a page forward to represent the `accountcreated.jsp` page. No data action is necessary to render this page because no bindings are used to display information. The page contains a single link that allows the user to return to the home page after creating their account:

```
<a href="<c:url value='home.do'/>">
             <bean:message key="accountcreated.gotomainpage"/></a>
```

In the following hands-on, you can optionally explore adding a databound text field to display the customer name in the page. This single change will necessitate changing the page forward to a data page element in the Struts diagram.

8.7 Hands-On for Lesson 5

The following hands-on shows how you can easily change a Struts page forward (`/accountcreated`) that displays no data bindings into an Oracle ADF data page capable of displaying the customer name in the page.

1. From the Struts page flow diagram, locate the **/accountcreated** page forward icon and double-click to open the `accountcreated.jsp` page in design view.
2. With the `accountcreated.jsp` page displayed, open the Data Control Palette and expand the **ToyStoreService** data control, **Accounts**.
3. Locate the **Firstname** attribute node under **Accounts** and drag the attribute node into the open page so that it appears before the message `accountcreated.header`.
4. In the Selected Page Flow Data Binding Option dialog, select **Convert the selected page to a data page** to convert the selected page to a data page and click **OK**.
5. Select the **Source** tab and locate the new value binding. The binding should appear before the `<bean:message>` tag. Type an extra space to separate the two:

```
<h2><c:out value="${bindings.Firstname}"/> <bean:message
                                               key="accountcreated.header"/>
</h2>
```

6. Return to the Struts page flow diagram and observe the new **/accountcreated** data page substituted for the original page forward. Select the **Source** tab and observe the modified action mapping:

```
<action path="/accountcreated"
        className="oracle.adf.controller.struts.actions.DataActionMapping"
        type="oracle.adf.controller.struts.actions.DataForwardAction"
        name="DataForm" parameter="/WEB-INF/jsp/accountcreated.jsp">
        <set-property property="modelReference"
                      value="WEB_INF_jsp_accountcreatedUIModel"/>
</action>
```

7. In the Application Navigator, expand **toystore.view** and double-click **WEB_INF_jsp_accountcreatedUIModel.xml** to open the newly created UI model definition file and observe the Oracle ADF data control definitions.

   The `UIModel.xml` definition file is created in JDeveloper the first time you drop a databound control from the Data Control Palette into your open JSP page.

8. In the Application Navigator, select **WEB_INF_jsp_accountcreatedUIModel.xml** so that it appears highlighted, and open the Structure window. Observe the **AccountIterator** iterator and **Firstname** value binding. You may double-click these items to edit the contents of the UI model definition file.

9. Right-click the **home.do** action and choose **Run** to launch the application. Click any category in the home page. Select any product link and add it to your cart. Proceed to checkout. When asked to sign in, click the **Register as a New User** link instead. Complete the form and supply a fictitious customer name and account information (be sure to observe validation errors for the entered data). Then click **Submit**. The customer's first name that you just created should appear in the account created page.

The data page manages the model data binding for the page. Oracle ADF provides the data page (`oracle.adf.controller.struts.actions.DataForwardAction`) to prepare the binding context for databound web pages and to execute custom business service methods exposed through the model. In this case, no business methods are required, and the standard `DataForwardAction` class will suffice to prepare the binding context before posting back to the page to be displayed.

Note that the binding displays the name in all lowercase. It is possible to create a service method to convert the `username` attribute to initial caps and to execute that method in a custom `DataForwardAction` class, similar to the ones described in previous lessons.

9 Lesson Six: Allowing the User to Edit Their Account

This chapter contains the following sections:

9.1 Introduction

In the previous lesson, we created the account object for a new user by generating a data entry form, using a generic, metadata-driven approach. In Lesson 6, we examine another way to display a data entry form, this time to edit user account information: the `editexistingaccount.jsp` page uses the traditional JSP page layout approach of placing each control inside an HTML form.

9.2 Setting Up the Model Layer Data

The `/EditAccountAction` data action sets up the model layer in its `initializeModelForPage()` method. It calls the custom service method `prepareToEditAccountInfoFor()` on the `ToyStoreService` interface, passing in the name of the current user as an argument. The implementation of this method in the `ToyStoreServiceImpl` class looks like this:

```
/* From: toystore.model.services.ToyStoreServiceImpl */
   public boolean prepareToEditAccountInfoFor(String username) {
     Key k = new Key(new Object[] { username });
     ViewObject vo = getAccounts();
     /*
      * We don't want the view object to execute any other query
      * than the one row we will be finding by key, so we mark
```

```
       * its max fetch size to zero.
       */
      vo.setMaxFetchSize(0);
      Row[] r = vo.findByKey(k, 1);
      if (r.length < 1) {
        return false;
      }
      Row rowFound = r[0];
      /*
       * Set the row we found as the current row in the VO
       */
      vo.setCurrentRow(rowFound);
      return true;
    }
```

The above service method performs the following three basic steps:

1. It creates an `oracle.jbo.Key` object based on the current user's name passed in.
2. It looks up an existing row in the `Accounts` view object by passing this key to the `findByKey()` method on the view object.
3. It sets that row as the current row in the view object.

9.3 Laying Out the Edit Account Page

In the corresponding JSP page, named `editexistingaccount.jsp`, we use the `<html:form>` tag from the Struts HTML tag library to implement the postback pattern by having its action post back to the data page like this:

```
<html:form action="editaccount.do" method="post">
```

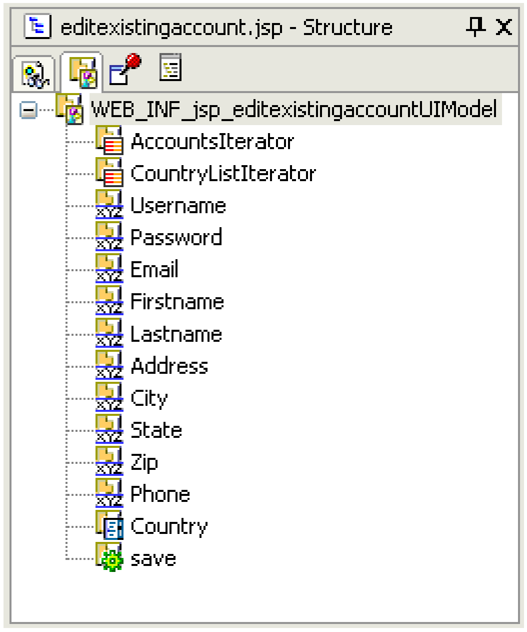
At runtime, the Struts `<html:form>` tag sees the action attribute value of `/updateaccount.do` and uses it, along with its action mapping information, to determine that the form bean named `DataForm` is the one that should be used to render this form. The `DataForm` form bean is defined in `struts-config.xml` to use our Oracle ADF `BindingContainerActionForm` class.

Since we're rendering the data entry form for just a single "row" of user account information, we don't need to use the JSTL `<c:forEach>` tag in this page and we don't need a range binding in our binding container. We simply format the individual fields in the form, using normal HTML table tags to get the prompts and controls to line up nicely. As this page shows off several different techniques in use, the following section will highlight each of the important ones.

9.4 Analyzing the Binding Container for the Edit Account Page

Figure 9–1, "The UI Model Tab Shows the Binding Container for the Edit Account Page", shows the binding container for the edit account page. Notice that we have basic attribute bindings for all of the `Accounts` object attributes except for `Country`, which is a list binding (its icon represents a poplist). We have two iterator bindings: `AccountsIterator` for the main account information we're editing, and `CountryListIterator` to supply a poplist with the valid country names from which the user can choose for the `Country` attribute.

Figure 9–1 The UI Model Tab Shows the Binding Container for the Edit Account Page



```
WEB_INF_jsp_editexistingaccountUIModel
    AccountsIterator
    CountryListIterator
    Username
    Password
    Email
    Firstname
    Lastname
    Address
    City
    State
    Zip
    Phone
    Country
    save
```

We also have an action binding named `save` that is bound to the built-in `Commit` operation on the `ToyStoreService` data control.

9.5 Showing Read-Only Data in a Form

The following sample shows the tags used by the page to output the HTML table row containing the prompt and data for the username field.

```
<%-- Username field --%>
<tr>
   <th align="right" title="<bean:message key="account.username.tooltip"/>">
     <bean:message key="account.username.label"/>
   </th>
   <td title="<bean:message key="account.username.tooltip"/>">
     <c:out value="${bindings.Username}"/>
   </td>
</tr>
```

Since `username` in this application is not updateable once it has been created, we don't need to render an HTML form control for the data. Using the `<c:out>` tag, we just output the value of the field for display using its corresponding binding object. The `<bean:message>` tags handle outputting translatable strings from the default `ToyStoreResources.properties` file to display the tooltip and label for the username.

9.6 Creating Input Fields in the Form

When data needs to be entered or edited, you can use a number of other tags in the Struts HTML library to render databound controls. The following code uses the `<html:password>` tag to show the `Password` property:

```
<%-- Password field --%>
<tr>
   <th align="right" title="<bean:message key="account.password.tooltip"/>">
```

```
      <bean:message key="dataentryform.mandatory"/>
      <bean:message key="account.password.label"/>
    </th>
    <td title="<bean:message key="account.password.tooltip"/>">
      <html:password property="Password" size="25" maxlength="30"/>
    </td>
    <td><html:errors property="Password"/></td>
</tr>
```

Recall that the Oracle ADF `BindingContainerActionForm` presents Struts (in this case, tags from the Struts HTML tag library) with a `DynaActionForm` bean having properties that are named for, and wired to, the bindings in your current binding container. So, when the `<html:password>` tag gets and sets the value of the `Password` property on this form bean, behind the scenes Oracle ADF is coordinating the properties of that form bean with the corresponding binding objects.

The above sample also illustrates using the Struts HTML tag `<html:errors>` to display any validation errors that are specific to the `Password` attribute. Of course, when the form is first rendered there won't be any validation errors, so this table cell will be empty. However, if the user submits the form and the model layer throws validation errors in, when this page is rendered again any errors related to the password will show up next to the password field on the screen.

Also, since the password field is mandatory, we've included a `<bean:write>` tag to show the string corresponding to the key `dataentryform.mandatory` as a visual marker for the user that the field is required. By default, we render an asterisk.

9.7 Using EL to Work with Labels, Tooltips, and Other Metadata

Oracle ADF entity object and view object components have a number of built-in features that allow developers to define control hints like locale-sensitive labels, tooltips, and format masks. The Oracle ADF binding layer exposes this metadata directly on the binding objects for convenient access by your view layer pages. The `<c:out>` tags shown below illustrate the EL expressions for the tooltip and label information that have been associated with the business object attributes or the view object attributes. If an entity object has defined a tooltip for one of its attributes named `Firstname`, for example, then this tooltip is inherited by any view objects that include `Firstname`. Of course, the view object can also override these control hints if necessary.

```
<%-- Firstname field --%>
<tr>
    <th align="right" title="<c:out value='${bindings.Firstname.tooltip}'/>">
      <c:if test="${bindings.Firstname.mandatory}">
        <bean:message key="dataentryform.mandatory"/>
      </c:if>
      <c:out value="${bindings.Firstname.label}"/>
    </th>
    <td>
      <html:text property="Firstname" size="30" maxlength="35"/>
    </td>
    <td><html:errors property="Firstname"/></td>
</tr>
```

Each binding object exposes runtime metadata about the objects to which it is bound that you can access at runtime using EL expressions. For example, the control value binding for an attribute exposes information about the underlying attribute in the model layer. The above sample shows an example of using this metadata to detect at runtime whether a given attribute, like `Firstname`, is mandatory or not. We can use this information, combined with the JSTL `<c:if>` tag, to conditionally output the mandatory marker on a required field.

```
<c:if test="${bindings.Firstname.mandatory}">
   <bean:message key="dataentryform.mandatory"/>
</c:if>
```

> **JDeveloper Tip:** To get a quick review of all the available properties on a binding object, just click on the binding in the **UI Model** tab of the Structure window and press the F1 key. The online help topic for the appropriate binding object appears in an IDE window for your reference.

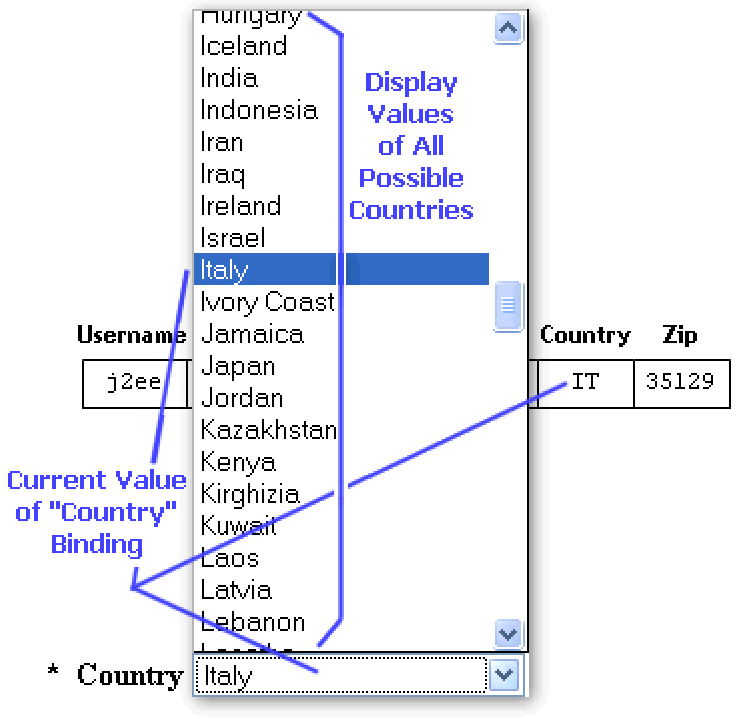
9.8 Including a Databound Poplist Control

Finally, we look at an example of a more sophisticated databound form control, such as a poplist showing the country of residence for a user. There are two dimensions to the poplist control:

- The value of the underlying `Country` binding, reflected by the selection in the list
- The list of all available country names to chose from

Figure 9–2, "The Poplist Displays the Current Value and Valid Values List", illustrates the poplist.

Figure 9–2 The Poplist Displays the Current Value and Valid Values List



Oracle ADF provides more sophisticated binding objects to handle controls like this one, which have multiple facets to their data binding requirements. The Oracle ADF list binding caters specifically to poplist-type controls that need to manage both a current bound attribute value and a list of valid choices to present to the user. Oracle ADF also supplies a tree binding object that is useful for displaying hierarchical data.

When you click the **UI Model** tab of the Structure window with **editexistingaccount.jsp** selected in the Application Navigator, you'll see the bindings described above. Now, select **CountryListIterator** and open the Property Inspector: you'll see that the **RangeSize** property has a value of `-1`. This value indicates that you want the range of the iterator to include all rows in the list of countries, rather than only a partial set.

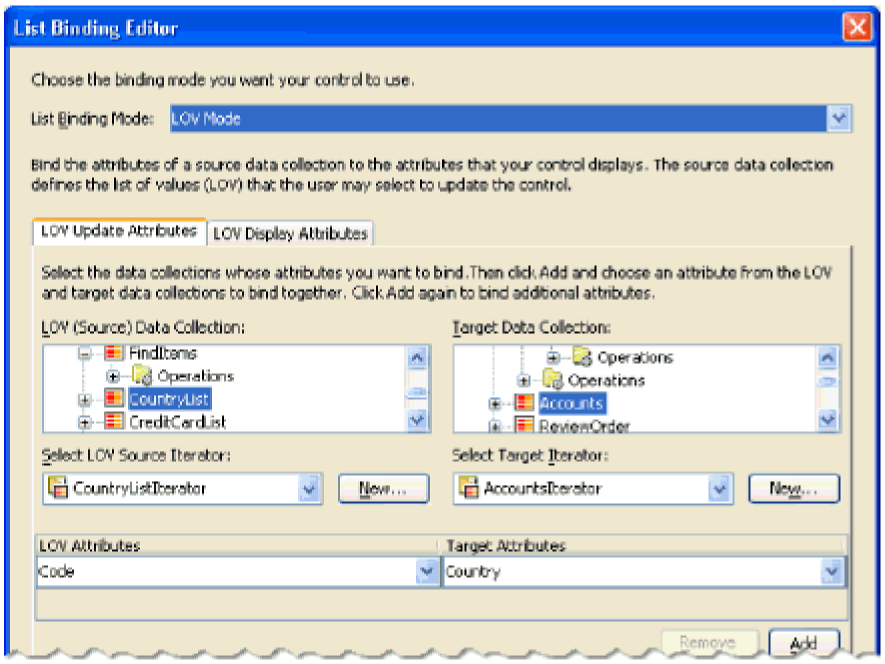
> **Best Practice Tip:** The iterator binding range size defaults to 10. For iterators driving the list of choices in a list binding, you will nearly always want to set the range size to be -1 as we've done here.

Right-click the **Country** binding and choose **Edit** from the context menu. You will see the List Binding Editor shown in the screenshot below. The editor allows you to see the binding metadata required to support the Country poplist:

- `CountryListIterator`, the datasource for the list of available choices.
- `AccountsIterator`, the iterator whose current row will be used both to determine the current value of the binding and to update the binding when the user selects a new item from the list.
- `Code` and `Country`, the source LOV and target attribute pairs, show that the value of the `Code` property from the selected row in `CountryList` will be set on the `Country` property on the current row of the target `AccountsIterator`.

If you click the **LOV Display Attributes** tab, you can observe that the `Description` attribute from the `CountryListIterator` is indicated as the value to be displayed to the user in the list. Figure 9–3, "The List Binding Editor for the Country List Binding", shows the editor.

Figure 9–3 The List Binding Editor for the Country List Binding


List Binding Editor

Choose the binding mode you want your control to use.

List Binding Mode: LOV Mode

Bind the attributes of a source data collection to the attributes that your control displays. The source data collection defines the list of values (LOV) that the user may select to update the control.

Select the data collections whose attributes you want to bind. Then click Add and choose an attribute from the LOV and target data collections to bind together. Click Add again to bind additional attributes.

Select LOV Source Iterator: CountryListIterator

Select Target Iterator: AccountsIterator

LOV Attributes	Target Attributes
Code	Country

The example below shows how to use `<html:select>` and `<html:optionsCollection>` to leverage the `Country` list binding and display the poplist in our page. The `<html:select>` element is bound to the `Country` property of the form bean, which corresponds to our list binding object. The

`<html:optionsCollection>` element gets its data from the nested, list-valued `displayData` property of that same `Country` binding. The beans in this display data collection each have a `prompt` and an `index` property, so in the following code we use those as the label and value (respectively) for each option in the list:

```
<%-- Country field --%>
<tr>
   <th align="right" title="<c:out value='${bindings.Country.tooltip}'/>">
     <c:if test="${bindings.Country.mandatory}">
       <bean:message key="dataentryform.mandatory"/>
     </c:if>
     <c:out value="${bindings.Country.label}"/>
   </th>
   <td>
     <html:select  property="Country" >
       <html:optionsCollection label="prompt"
       value="index"
       property="Country.displayData" />
     </html:select>
   </td>
   <td><html:errors property="Country"/></td>
</tr>
```

Note: For bandwidth optimization, the Oracle ADF binding layer expects the nonvisible values of a list binding to be the zero-based `index` number in their `displayData` collection. The Oracle ADF list binding handles translating the underlying Country value (like `IT`, for example) into an index position (like `86`) in the list of values on both read and write of the binding value.

9.9 Alternative to a Databound Poplist Using Custom Renderer

In contrast to the above approach, which relied on the control to render the poplist in the form, we can accomplish the same thing in a more generic, metadata-driven way by performing these steps:

1. Create a custom renderer class to determine how to handle the country list.
2. Set a custom property on the attribute of the Oracle ADF Business Components to specify the custom renderer.
3. Work with the `<adf:inputrender>` tag in the JSP page to apply the custom renderer.

Note: The `<adf:inputrender>` tag implementation consults the value of the `EditRenderer` attribute property to determine whether the attribute has specified a custom renderer. If none is specified, Oracle ADF picks an appropriate control to render the data.

This use of the custom renderer is demonstrated by the `formControl.jsp` page introduced in Lesson 5, and described in more detail below.

In the case of the `formControl.jsp` page, we've specified the class name `toystore.fwk.view.ListBindingPoplistRenderer`, which implements a customized poplist renderer for Oracle ADF list bindings. This custom field renderer extends the default `oracle.jdeveloper.html.StaticPickList` renderer to populate some of its properties based on information it can retrieve from the list binding object. The source code for the custom renderer (from the `FwkExtensions` project) is shown below:

```
package toystore.fwk.view;
import java.util.List;
import java.util.Map;
import oracle.jbo.Row;
import oracle.jbo.html.BindingContainerDataSource;
import oracle.jbo.uicli.binding.JUControlBinding;
import oracle.jbo.uicli.binding.JUCtrlListBinding;
import oracle.jdeveloper.html.StaticPickList;
/**
  * Extends the oracle.jdeveloper.html.StaticPickList renderer to drive
  * off of a list binding.
  */
public class ListBindingPoplistRenderer extends StaticPickList {
   /**
    * Overrides renderToString() in StaticPickList
    */
   public String renderToString(Row row) {
     BindingContainerDataSource ds = (BindingContainerDataSource)getDatasource();
     JUControlBinding b = ds.getControlBinding();
     String[] labels = null;
     String[] values = null;
     if (b instanceof JUCtrlListBinding) {
       JUCtrlListBinding listBinding = (JUCtrlListBinding)b;
       List valueList = listBinding.getDisplayData();
       int size = valueList.size();
       values = new String[size];
       labels = new String[size];
       for (int z = 0; z < size; z++) {
         labels[z] = (String)((Map)valueList.get(z)).get("prompt");
         values[z] = Integer.toString(z);
       }
       setValue(Integer.toString(listBinding.getSelectedIndex()));
     }
     setDataSource(labels,values);
     return super.renderToString(row);
   }
}
```

You can see that the code accesses the `JUControlBinding` object from the datasource, that it checks to be sure the object is a `JUCtrlListBinding` object, and that it calls the `getDisplayData()` method on the list binding to access the list display data. In order to populate the `String[]` variables for the labels and the values, it iterates over the display data collection and adds the `prompt` attribute from each bean in the collection to the `label` array. Since the Oracle ADF binding layer will expect the value returned from the page to be the numerical row number, we populate the `values` array by converting the loop variable `z` to a string on each iteration. The net effect is that when our generic `formControl.jsp` "component" page renders an HTML form for the bindings in the current binding container, the `Country` binding will be rendered as a databound poplist populated from the display data collection named `CountryList`.

If you adopt a generic data form rendering technique like the one employed by the `formControl.jsp` page in your applications, you can more easily ensure that all data entry forms in your application look and act similarly.

9.10 Hands-On for Lesson 6

The following hands-on shows how changing the iterator `rangesize` property controls the display range.

1. In the Application Navigator, locate **editexistingaccount.jsp** in the **WEB-INF/jsp** folder and select it so that it appears highlighted.
2. Open the Structure window and select the **UI Model** tab to view the list of binding definitions in the **WEB_INF_jsp_editexistingaccountUIModel** definition file.
3. In the Structure window, double-click **CountryListIterator** and observe that **CountryListIterator** is bound to the **CountryList** data collection. Click **Cancel** to exit the editor.
4. With **CountryListIterator** selected, open the Property Inspector and locate the **RangeSize** property with the value `-1`.
5. In the Property Inspector, change the **RangeSize** property value to `10` and press Enter. This will limit the display list to just ten objects from the bound data collection.
6. Right-click the **home.do** action and choose **Run** to launch the application. Click the **Login** icon and enter the `J2EE/J2EE` username and password. Click the **Edit Account** icon and display the dropdown list for the **Country** field. The list should be limited to just ten rows.

The iterator binding is a runtime object that your application creates to access the Oracle ADF binding context. The iterator binding holds references to the bound data collection, it accesses the collection, and it iterates over its data objects. You can set the number of data objects to be fetched from the bound data collection, such that only the number is displayed on the page. The range you specify defines a window you can use to access a subset of the data objects in the collection. By default, the range size is set to just ten data objects.

10 Summary of the Oracle ADF Toy Store Application

The Oracle ADF Toy Store application, built using Oracle ADF and Apache Struts, was created to show a realistic, functional Oracle ADF-based web application. We've seen that the Java coding needed is minimal and that, when required, it is code that is focused directly on the business application problem at hand. Additionally, both Struts and Oracle ADF make extensive use of XML-based configuration information for their framework components, which further simplifies the design of the web pages by driving a lot of framework behavior from this metadata, rather than from a heavy code generation.

In the lessons of this case study, we analyzed many of the key aspects of the Oracle ADF Toy Store application to better understand how a web application like this can adhere to an MVC architecture. Along the way, we noted numerous Oracle JDeveloper 10*g* features for simplifying the development of our model, view, and controller layer components.

JDeveloper offers integrated support for:

- Binding your user interfaces to any kind of business service data, using a consistent, visual, and declarative approach
- Creating, configuring, and evolving every aspect of your web application through synchronized design time tools like the Application Navigator, the Structure window, the Property Inspector, wizards, and object editors
- Visually editing your Struts page flow, so as to more easily understand and modify all of the configuration information in your `struts-config.xml` file
- Visually designing your JSP pages, so as to more easily create web pages

10.1 Related Documentation

- `Oracle Application Development Framework FAQ`, a brief overview posted on the Oracle Technology Network (OTN)
- `Building a Web Store with Apache Struts and Oracle ADF Frameworks`, a technical whitepaper on OTN
- `Oracle ADF Data Binding Primer and Struts Integration Overview`, a technical whitepaper on OTN
- `Oracle Application Development Framework Development Guidelines Manual`, a PDF document available from the JDeveloper page on OTN