Sun™ ONE Studio 4, Community Edition Tutorial

Sun Microsystems, Inc.
4150 Network Circle
Santa Clara, CA 95054 U.S.A.
650-960-1300

Part No. 816-7868-10
December 2002 Revision A

Send comments about this document to: docfeedback@sun.com

Contents

Figures

Tables

Before You Begin

Welcome to the Sun™ Open Net Environment (Sun ONE) Studio 4, Community Edition tutorial. In this tutorial, you will learn how to use the features introduced in the Community Edition, namely:

- Support for web applications that use Java™ Servlet and JavaServer Pages™ (JSP™) technology
- Database access using the JSP Standard Tag Library (JSTL) reference implementation from the Jakarta Project

See the release notes for a list of environments in which you can create the examples in this book. The release notes are available on this web page:

`http://forte.sun.com/ffj/documentation/index.html`

Screen shots vary slightly from one platform to another. You should have no trouble translating the slight differences to your platform. Although almost all procedures use the interface of the Sun™ ONE Studio 4 software, occasionally you might be instructed to enter a command at the command line. Here too, there are slight differences from one platform to another. For example, a Microsoft Windows command might look like this:

```
c:>cd MyWorkDir\MyPackage
```

To translate for UNIX® or Linux environments, simply change the prompt and use forward slashes:

```
% cd MyWorkDir/MyPackage
```

Before You Read This Book

This tutorial creates a simple web application that interacts with a database and displays dynamically generated content. The design and architecture conforms to the Java 2 Platform, Enterprise Edition (J2EE™) Blueprints. If you want to learn how to use the features of Sun ONE Studio 4, Community Edition to build the components of a web application, you will benefit from working through this tutorial.

Before starting this tutorial, you should be familiar with the following subjects:

- Java programming language
- Java Servlet syntax
- JDBC™ enabled driver syntax
- JavaServer Pages syntax
- HTML syntax
- JSP Standard Tag Library (JSTL) from the Jakarta Project
- Relational database concepts (such as tables and keys)
- How to use the chosen database

This book requires a knowledge of Java Servlet and JavaServer Pages concepts, including web applications. The following resources define these concepts:

- *Java Servlet Specification Version 2.3*
  `http://java.sun.com/products/servlet/download.html#specs`
- *JavaServer Pages Specification Version 1.2*
  `http://java.sun.com/products/jsp/download.html#specs`

For the JSTL tutorial, as well as links to other useful information, see:
`http://jakarta.apache.org/taglibs/tutorial.html`

Note – Sun is not responsible for the availability of third-party web sites mentioned in this document and does not endorse and is not responsible or liable for any content, advertising, products, or other materials on or available from such sites or resources. Sun will not be responsible or liable for any damage or loss caused or alleged to be caused by or in connection with use of or reliance on any such content, goods, or services available on or through any such sites or resources.

How This Book Is Organized

This manual is designed to be read from beginning to end. Each chapter in the tutorial builds upon the code developed in earlier chapters.

Chapter 1 describes the software requirements for the CDShopCart tutorial, explains how to install the tutorial database tables, shows how to start the Sun ONE Studio 4 integrated development environment (IDE), and how to verify that the IDE is using the correct web server. This chapter includes a descriptive list of the installed Sun ONE Studio 4 directories.

Chapter 2 describes the architecture of the CDShopCart application.

Chapter 3 provides step-by-step instructions for creating the CDShopCart application, a simple online shopping cart application for the purchase of music CDs.

Appendix A provides complete source files for the tutorial application.

Appendix B provides database script files for the tutorial application.

Typographic Conventions

Typeface	Meaning	Examples
`AaBbCc123`	The names of commands, files, and directories; on-screen computer output	Edit your `.cvspass` file. Use `DIR` to list all files. `Search is complete.`
`AaBbCc123`	What you type, when contrasted with on-screen computer output	`> login` `Password:`
AaBbCc123	Book titles, new words or terms, words to be emphasized	Read Chapter 6 in the *User's Guide.* These are called *class* options. You *must* save your changes.
AaBbCc123	Command-line variable; replace with a real name or value	To delete a file, type **`DEL`** *filename.*

Related Documentation

Sun ONE Studio 4 documentation includes books delivered in Acrobat Reader (PDF) format, release notes, online help, readme files for example applications, and Javadoc™ documentation.

Documentation Available Online

The documents described in this section are available from the `docs.sun.com`SM web site and from the documentation page of the Sun ONE Studio Developer Resources portal (`http://forte.sun.com/ffj/documentation`).

The `docs.sun.com` web site (`http://docs.sun.com`) enables you to read, print, and buy Sun Microsystems manuals through the Internet. If you cannot find a manual, see the documentation index installed with the product on your local system or network.

- Release notes (HTML format)

  Available for each Sun ONE Studio 4 edition. Describe last-minute release changes and technical notes.

- Getting Started guides (PDF format)

  Describe how to install the Sun ONE Studio 4 integrated development environment (IDE) on each supported platform and include other pertinent information, such as system requirements, upgrade instructions, application server information, command-line switches, installed subdirectories, database integration, and information on how to use the Update Center.

  - *Sun ONE Studio 4, Community Edition Getting Started Guide* - part no. 816-7871-10
  - *Sun ONE Studio 4, Enterprise Edition for Java Getting Started Guide* - part no. 816-7859-10
  - *Sun ONE Studio 4, Mobile Edition Getting Started Guide* - part no. 816-7872-10

- Sun ONE Studio 4 Programming series (PDF format)
  - This series provides in-depth information on how to use various Sun ONE Studio 4 features to develop well-formed J2EE applications.*Building Web Components* - part no. 816-7869-10

    Describes how to build a web application as a J2EE web module using JSP pages, servlets, tag libraries, and supporting classes and files.

  - *Building J2EE Applications* - part no. 816-7863-10

Describes how to assemble EJB modules and web modules into a J2EE application, and how to deploy and run a J2EE application.

- *Building Enterprise JavaBeans Components* - part no. 816-7864-10

  Describes how to build EJB components (session beans, message-driven beans, and entity beans with container-managed or bean-managed persistence) using the Sun ONE Studio 4 EJB Builder wizard and other components of the IDE.

- *Building Web Services* - part no. 816-7862-10

  Describes how to use the Sun ONE Studio 4 IDE to build web services, to make web services available to others through a UDDI registry, and to generate web service clients from a local web service or a UDDI registry.

- *Using Java DataBase Connectivity* - part no. 816-7870-10

  Describes how to use the JDBC productivity enhancement tools of the Sun ONE Studio 4 IDE, including how to use them to create a JDBC application.

- Sun ONE Studio 4 tutorials (PDF format)

  These tutorials demonstrate how to use the major features of each Sun ONE Studio 4 edition.

  - *Sun ONE Studio 4, Community Edition Tutorial* - part no. 816-7868-10

    Provides step-by-step instructions for building a simple J2EE web application.

  - *Sun ONE Studio 4, Enterprise Edition for Java Tutorial* - part no. 816-7860-10

    Provides step-by-step instructions for building an application using EJB components and Web Services technology.

  - *Sun ONE Studio 4, Mobile Edition Tutorial* - part no. 816-7873-10

    Provides step-by-step instructions for building a simple application for a wireless device, such as a cellular phone or personal digital assistant (PDA). The application will be compliant with the Java 2 Platform, Micro Edition (J2ME™ platform) and conform to the Mobile Information Device Profile (MIDP) and Connected, Limited Device Configuration (CLDC).

You can also find the completed tutorial applications at:
`http://forte.sun.com/ffj/documentation/tutorialsandexamples.html`

Online Help

Online help is available inside the Sun ONE Studio 4 IDE. You can open help by pressing the help key (F1 in Microsoft Windows and Linux environments, Help key in the Solaris environment), or by choosing Help → Contents. Either action displays a list of help topics and a search facility.

Examples

You can download examples that illustrate a particular Sun ONE Studio 4 feature, as well as completed tutorial applications, from the Sun ONE Studio Developer Resources portal at:

`http://forte.sun.com/ffj/documentation/tutorialsandexamples.html`

The site includes the applications used in this document.

Javadoc Documentation

Javadoc documentation is available within the IDE for many Sun ONE Studio 4 modules. Refer to the release notes for instructions on installing this documentation. When you start the IDE, you can access this Javadoc documentation within the Javadoc pane of the Explorer.

Sun Welcomes Your Comments

Sun is interested in improving its documentation and welcomes your comments and suggestions. Email your comments to Sun at this address:

`docfeedback@sun.com`

Please include the part number (816-7868-10) of your document in the subject line of your email.

CHAPTER 1

Getting Started

This chapter explains what you must do before starting the Sun ONE Studio 4, Community Edition tutorial. For your convenience, it duplicates some installation information from the *Sun ONE Studio 4, Community Edition Getting Started Guide*. The topics covered in this chapter are:

Note – There are several references in this book to the *CDShopCart application files*. These files include a completed version of the tutorial application, a Readme file describing how to run the completed application, and several versions of the SQL script for creating the required database table. These files are compressed into a ZIP file you can download from the Sun ONE Studio 4 Developer Resources portal at `http://forte.sun.com/ffj/documentation/tutorialsandexamples.html`

Software Requirements for the Tutorial

This section describes how to prepare your system before starting the Sun ONE Studio 4, Community Edition tutorial. This means making sure you have everything required to run the Sun ONE Studio 4 integrated development environment (IDE), as well as what is required to create and run the tutorial.

You can access general system requirements from the release notes or from the Sun ONE Studio 4 Developer Resources portal's Documentation page at `http://forte.sun.com/ffj/documentation/`.

What You Need to Run the Sun ONE Studio 4 IDE

The Sun ONE Studio 4 IDE requires the Java™ 2 Software Development Kit (the J2SE™ SDK). When you install the IDE, the installer searches your system for the J2SE SDK software and will notify you and stop the installation if the correct version is not installed on your system. You can download the correct version of the J2SE SDK from the Java Developer's portal at `http://java.sun.com/j2se/`.

What You Need to Create and Run the Tutorial

You need the following items to create and run the tutorial. Some of these items are included in the default installation of Sun ONE Studio 4, Community Edition. For the supported versions of these products, see the release notes on the following web site: `http://forte.sun.com/ffj/documentation/index.html`

- Database software—any of the following:
  - PointBase Server

    You can install PointBase when you install Sun ONE Studio 4, Community Edition. To see whether PointBase was installed, look for a `pointbase` directory under the directory where Sun ONE Studio 4 is installed. If no `pointbase` directory is there, you can run the installer again to install it.
  - Oracle 8.1.7, JDBC Thin Driver 8.1.7
  - Microsoft SQL Server 2000, WebLogic JDBC driver 5.1.0 or JDBC-ODBC bridge (SQL Server 2000 ODBC driver)
  - IBM DB2 7.1, JDBC Thin Driver for DB2 7.1
- The SQL script that creates the tutorial database table

  The tutorial SQL scripts are provided in Appendix B. They are also included in the application files of the CDShopCart tutorial, available from the Sun ONE Studio 4 Developer Resources portal. See "Creating the Tutorial Database Table" on page 25 for information on installing the tutorial database table.
- A web server

  The tutorial is a web application, which requires a web server. This tutorial uses an embedded version of Tomcat, version 4.0, within the IDE that provides the functionality of a web server for testing purposes.
- A web browser

  You need a web browser to view the tutorial application pages. This can be either Netscape Communicator™ or Microsoft's Internet Explorer.

Starting the Sun ONE Studio 4 IDE

Start the Sun ONE Studio 4 IDE by running the program executable, as described in the following sections, and more fully in the *Sun ONE Studio 4, Community Edition Getting Started Guide.*

Starting the IDE on Microsoft Windows

After installation, start the IDE in one of the following ways:

- **Double-click the Sun ONE Studio 4 CE icon on your desktop.**

  This runs the `runidew.exe` executable, which launches the IDE without a console window. This executable exists in the *s1studio-home*`\bin` directory, along with an alternative executable, `runide.exe`. The `runide.exe` icon launches the IDE with a console window that includes standard error and standard output from the IDE. On the console, you can press Ctrl-Break to get a list of running threads or Ctrl-C to immediately terminate the program.

- **Choose Start → Programs → Sun Microsystems → Sun ONE Studio 4 CE → Sun ONE Studio**

- **Run any of the executables from the command line.**

  For example:

```
C:\> s1studio-home\bin\runide.exe [switch]
```

For information on switches, see “Modifying the Session With Command-Line Switches” on page 20.

Starting the IDE on Solaris, UNIX, and Linux Environments

After installation, a `runide.sh` script is in *s1studio-home*`/bin` directory. Launch this script by typing the following in a terminal window:

```
$ sh s1studio-home/bin/runide.sh
```

Modifying the Session With Command-Line Switches

TABLE 1-1 describes the switches that you can use to modify how you launch the IDE. This information is also available from the *Sun ONE Studio 4, Community Edition Getting Started Guide*, but is provided here for your convenience.

- On Microsoft Windows systems

  You can set options when running the IDE on the command line.

- In Solaris, UNIX, and Linux environments

  You can modify the `ide.sh` file in the `bin` subdirectory of the installation directory, or you can create your own shell script that calls `ide.sh` with options.

TABLE 1-1 `runide` Command-Line Switches

Switch	Meaning
`-classic`	Uses the classic JVM.
`-cp:p` *addl-classpath*	Adds a class path to the beginning of the Sun ONE Studio 4 class path.
`-cp:a` *addl-classpath*	Adds a class path to the end of the Sun ONE Studio 4 class path
`-fontsize` *size*	Sets the font size used in the GUI to the specified size.
`-locale` *language* [:*country*[:*variant*]]	Uses the specified locale for the session instead of the default locale.
`-J`*jvm-flags*	Passes the specified flag directly to the JVM. (There is no space between `-J` and the argument.)
`-jdkhome` *jdk-home-dir*	Uses the specified Java 2 SDK instead of the default SDK.
`-h` or `-help`	Opens a GUI dialog box that lists the command-line options.
`-hotspot` or `-client` or `-server` or `-classic` or `-native` or `-green`	Uses the specified variant of JVM.

TABLE 1-1 `runide` Command-Line Switches *(Continued)*

Switch	Meaning
`-single`	Runs the IDE in single-user mode. Enables you to launch the IDE from *s1studio-home* instead of from your user settings directory.
`-ui` *UI-class-name*	Runs the IDE with the given class as the IDE's look and feel.
`-userdir` *user-directory*	Uses the specified directory for your user settings for the current session. See the next section for more information.

Specifying Your User Settings Directory

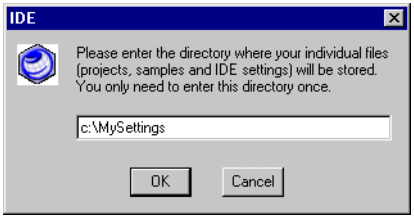
The Sun ONE Studio 4 IDE stores your individual projects, samples, and IDE settings in your own special directory. This enables individual developers to synchronize their development activities, while keeping their own personal work and preferences separate.

- In Solaris, UNIX, or Linux environments

  If you don't explicitly specify a user settings directory with the `-userdir` command-line switch, user settings are located by default in *user-home*`/ffjuser40ce`.

- On Microsoft Windows systems

  At first launch of the Sun ONE Studio 4 IDE, you are prompted to specify a user settings directory. Use a complete specification, for example, `C:\MySettings`.



  This value is stored in the registry for later use. For a given session, you can specify a different user settings directory by using the `-userdir` command-line switch when launching the IDE.

Understanding the Sun ONE Studio 4 Directory Structure

When you install the Sun ONE Studio 4 software, the subdirectories described in TABLE 1-2 are included in your installation directory.

TABLE 1-2 Sun ONE Studio 4 Directory Structure

Directory	Purpose
_uninst	Contains the files used to uninstall the IDE.
beans	Contains JavaBeans™ components installed in the IDE.
bin	Includes Sun ONE Studio 4 launchers (as well as the ide.cfg file on Microsoft Windows installations). (Solaris only) Also contains launch points for the standalone applications delivered with the Solaris tools, including xemacs, gvim, xdesigner, and forte_fcc.
docs	Contains the Sun ONE Studio 4 help files and other miscellaneous documentation.
emacs	(Solaris only) Contains emacs files.
examples	Contains source files for examples that illustrate several key features of the Sun ONE Studio 4, Enterprise Edition for Java.
j2sdkee1.3.1	(English-only installation) Contains the Java 2 Platform, Enterprise Edition (J2EE™ SDK) v. 1.3.1, if you chose to install it during the Sun ONE Studio 4 installation process.
lib	Contains JAR files that make up the IDE's core implementation and the open APIs.
man	(Solaris only) Contains man pages for Solaris Developer Modules, if installed.
modules	Contains JAR files of Sun ONE Studio 4 modules.
pointbase	(English-only installation) Contains the executables, classes, databases, and documentation for the PointBase Server 4.2 Restricted Edition database (if installed).

TABLE 1-2 Sun ONE Studio 4 Directory Structure *(Continued)*

Directory	Purpose
`sources`	Contains sources for libraries that might be redistributed with user applications.
`system`	Includes files and directories used by the IDE for special purposes. Among these are `ide.log`, which provides information useful when seeking technical support.
`tomcat401`	Contains sources for the Tomcat, 4.01, web server.

When you launch the Sun ONE Studio 4 software, the subdirectories in TABLE 1-3 are installed in your user settings directory. Most of them correspond to subdirectories in the Sun ONE Studio 4 home directory, and are used to hold your settings.

TABLE 1-3 Directory Structure for the User Settings Directory

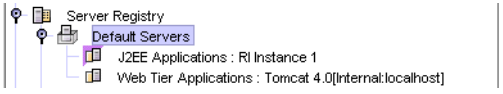
Directory	Purpose
`beans`	Contains user settings for JavaBeans components installed in the IDE.
`javadoc`	Contains user settings for Javadoc files installed in the IDE.
`j2sdkee1.3.1`	Contains the Java 2 Platform, Enterprise Edition (J2EE™ SDK) v. 1.3.1, if you chose to install it during the Sun ONE Studio 4 installation process.
`lib`	Contains user settings for the system `lib` files.
`modules`	Contains modules downloaded from the Update Center.
`pointbase`	Contains the executables, classes, databases, and documentation for the PointBase Server 4.2 Restricted Edition database (if installed).
`sampledir`	The directory mounted by default in the Filesystems pane of the Explorer. Objects you create in the IDE are saved here unless you mount other directories and use them instead.
`sampledir/examples`	Contains several NetBeans example applications.
`system`	Contains user settings for system files and directories.
`tomcat401_base`	Contains user settings for your work with JSP pages.

Verifying the Correct Default Web Server

The CDShopCart tutorial uses the Tomcat 4.0 web server. This server is installed and set as the default web server by the Sun ONE Studio 4 installer. However, you should verify that Tomcat is the default server before you test or run the CDShopCart application.

To verify that the Tomcat web server is the default server:

1. **In the IDE, click the Explorer's Runtime tab.**
2. **Expand the** `Server Registry` **node and its** `Default Servers` **subnode.**
   - If the `Default Servers` node looks like this, then the IDE is using the correct server.



   - If anything other than Tomcat 4.0 is listed, then:

   a. **Right-click the default web server node and choose Set Default Server.**

      The Select Default Web Server dialog box is displayed.

   b. **Select the correct server and click OK.**

Creating the Tutorial Database Table

Before you start the Sun ONE Studio 4, Community Edition tutorial, you must create and install a database table. Instructions in this section describe how to install the table in a PointBase database. If you wish to use a different database program, other versions of the tutorial SQL script are provided in Appendix B.

Alternatively, files containing the SQL script are included in the zip file of the tutorial source code, which you can download from the Sun ONE Studio Developer Resources portal at:

`http://forte.sun.com/ffj/documentation/tutorialsandexamples.html`

TABLE 1-4 identifies the four available SQL script files.

TABLE 1-4 SQL Scripts for Creating the Tutorial Table

Script name	Description
`CDCatalog_pb.sql`	Creates and populates the table used by the tutorial in PointBase SQL format. Contents are in “Script for a PointBase Database” on page 94.
`CDCatalog_ora.sql`	Creates and populates the table used by the tutorial in Oracle SQL format. Contents are in “Script for an Oracle Database” on page 95.
`CDCatalog_ms.sql`	Creates and populates the table used by the tutorial in Microsoft SQLServer SQL format. Contents are in “Script for a Microsoft SQLServer Database” on page 96.
`CDCatalog_db2.sql`	Creates and populates the table used by the tutorial in IBM DB2 SQL format. Contents are in “Script for an IBM DB2 Database” on page 97.

The `CDCatalog` script creates the database schema shown in TABLE 1-5.

TABLE 1-5 `CDCatalog` Database Table

Table Name	Columns	Primary Key	Other
CD	`id`	yes	
	`cdtitle`		
	`artist`		
	`country`		
	`price`		

The CD table is populated with the records shown in TABLE 1-6.

TABLE 1-6 CD Table Records

ID	CDtitle	Artist	Country	Price
1	Yuan	The Guo Brothers	China	14.95
2	Drums of Passion	Babatunde Olatunji	Nigeria	16.95
3	Kaira	Tounami Diabate	Mali	13.95
4	The Lion is Loose	Eliades Ochoa	Cuba	12.95
5	Dance the Devil Away	Outback	Australia	14.95

First create a tutorial database and load the table into it. (If you prefer, you can install the table in any PointBase database you choose.)

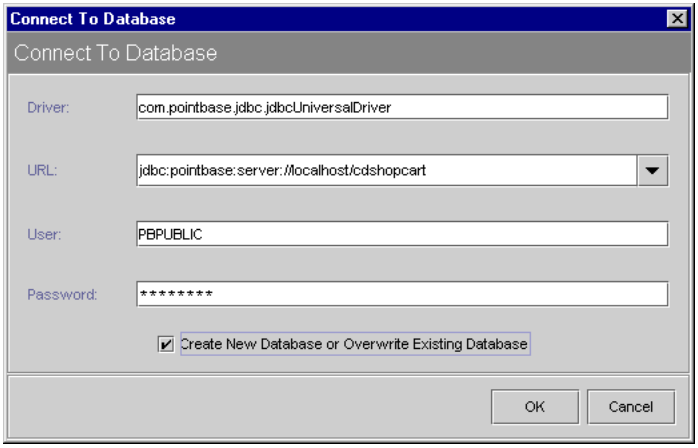
To create the tutorial database table:

1. **Start the IDE, if you have not already started it.**
2. **Start the PointBase Server by choosing Tools → PointBase Network Server → Start Server.**

   A PointBase Server dialog box is displayed. Minimize the dialog box window.
3. **Start the PointBase Console.**
   - In Solaris or Linux environments: Run the `Console` file in the *s1studio-home*`/pointbase/client` directory.
   - On Microsoft Windows: Choose Start → Sun Microsystems → Sun ONE Studio 4 CE → PointBase → Console or double-click the `console.bat` file in the *s1studio-home*`/pointbase/client` directory.

     The Connect To Database dialog box appears, showing values for the PointBase driver to the default sample database.

4. **Change the word `sample` at the end of the URL field to `cdshopcart`, as shown.**



5. **Set the Create New Database option and click OK.**

   The PointBase Console is displayed. Wait until the status message ending in "Ready" is displayed before proceeding.

6. **Copy the PointBase script from "Script for a PointBase Database" on page 94 and paste it into the SQL entry window of the Console.**

   Alternatively, if you have the `CDCatalog_pb.sql` file from the tutorial source zip file, do this:

   a. **Choose File → Open to display the file browser dialog box.**

   b. **Use the file browser to find the `CDCatalog_pb.sql` file and click Open.**

7. **Choose SQL → Execute All.**

   The message window confirms that the script was executed. (Ignore the initial messages beginning "Cannot find the table..." These appear because there is a DROP statements for the table, which has not been created yet. This DROP statement will be useful in the future if you want to rerun the script to initialize the table.)

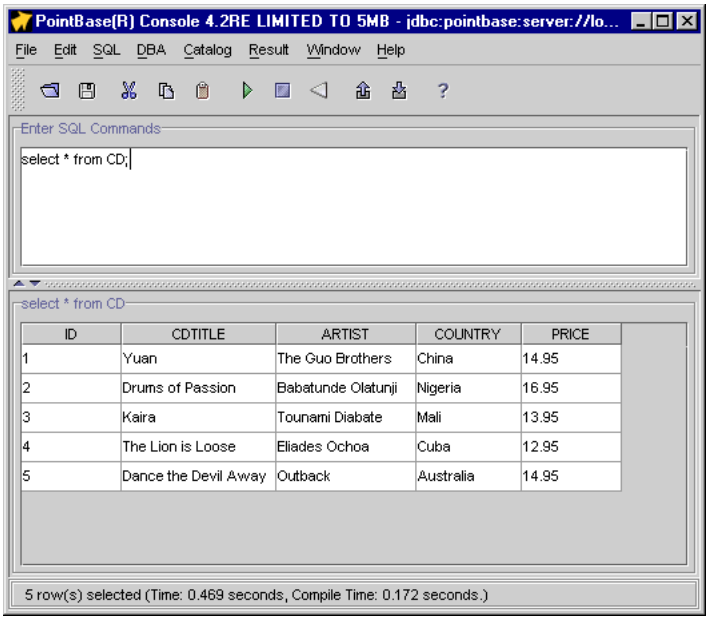
   Two database files are created in the *s1studio-home*`/pointbase/databases` directory.

8. **Test that you have created the table by clearing the SQL entry window (Window → Clear Input) and typing:**

```
SELECT * FROM CD;
```

9. **Choose SQL → Execute.**

   Your console should display the CD table.



Note – If your display does not look like this table, choose Window → Windows to change the display type.

---

10. **Close the PointBase Console window.**

    Now, you are ready to start the tutorial.

CHAPTER 2

Introduction to CDShopCart

In the process of creating the tutorial example application, you learn how to build components of a web application with Sun ONE Studio 4, Community Edition features.

This chapter describes the application you will build, first laying out its requirements and then describing an architecture that will fulfill those requirements. The final section describes how you use Sun ONE Studio 4, Community Edition features—web module constructs and Sun ONE Studio 4 tag libraries—to create the application.

This chapter is organized into the following sections:

Functionality of the Tutorial Application

The tutorial example application, CDShopCart, is a simple online shopping cart application for the purchase of music CDs. The customer uses a web browser to interact with the application's interface, as follows:

1. The customer selects items from a catalog page to add to a shopping cart.
2. The customer can add more items from the catalog page or remove existing items from the shopping cart.
3. When the customer is ready to make a purchase, the application displays a Thank You message for the order and ends the session.

4. The customer can then exit the application or return to the order page to start a new shopping session.

Application Scenarios

The interaction of the CDShopCart application begins with the customer's visit to the application's catalog page and ends when the customer either completes an order or quits the site. The following scenarios describe a few of the customer's interactions with the CDShopCart application. Walking through these scenarios illustrates the requirements of the application, as well as interactions that happen within the application.

1. The customer starts the application by pointing the browser to the URL of the application's home page.

   The home page is the CD Catalog List page, which displays a list of available music CDs and their associated information: title, the CD id number, the name of the performing artist, the artist's country, and the price.

2. The customer selects a CD for purchase by clicking the Add button associated with a CD.

   This action causes the application to display a Shopping Cart page showing the selected CD title, its ID number, and price.

3. The customer selects more CDs for purchase.

   The customer clicks the Resume Shopping button on the Shopping Cart page, which causes the application to redisplay the CD catalog page for further selections. The customer can repeat this sequence as many times as she likes, even adding the same CD multiple times (which adds more rows of the same CD to the cart—there is no Amount column for each CD).

4. The customer removes an item from her shopping cart by clicking the Delete button associated with the item on the Shopping Cart page.

   Clicking this button redisplays the shopping cart minus the item, unless she deleted the last CD in the cart. When the last CD is removed, a page is displayed that reports that the cart is empty.

5. The customer can click the Resume Shopping button on the page to return to the CD Catalog List page, or the Cancel Order button to end the session. (The Cancel Order page is discussed in Scenario 7.)

6. The customer decides to make a purchase and clicks the Place Order button on the Shopping Cart page.

This action displays a Thank You message and ends the session. The customer can click the Resume Shopping link on the message page to start another session, or leave the application by closing the browser or going to a different URL.

7. The customer cancels an order at any time by clicking the Cancel Order button on the Shopping Cart page.

   This causes the application to display a Cancel Order message page that ends the session. The Cancel Order page includes a Resume Shopping button, in case the customer wants to start a new session.

Application Functional Specification

Given the kinds of scenarios in which the CDShopCart application would be used, the following items list the main functions for a user interface of an application that supports these shopping interactions.

- A set of links to navigate from page to page
- A master view of all the site's offerings through a displayed list
- A view of items selected for purchase
- Buttons on the catalog page for adding each item
- Buttons on the shop cart page for removing items
- A button on the shop cart page to initiate checkout
- A button on the shop cart page to cancel the order
- A button on the checkout page to return to the home page to begin a new order
- A button on the empty cart page to return to the home page
- A button on the empty cart page to cancel the order

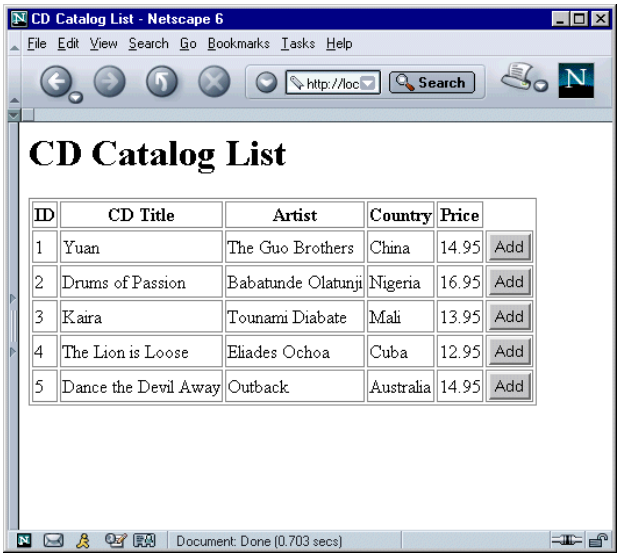
User's View of the Tutorial Application

The user's view of the application illustrates how the scenarios and the functional specification, described in "Functionality of the Tutorial Application" on page 29 are realized.

To run the CDShopCart application:

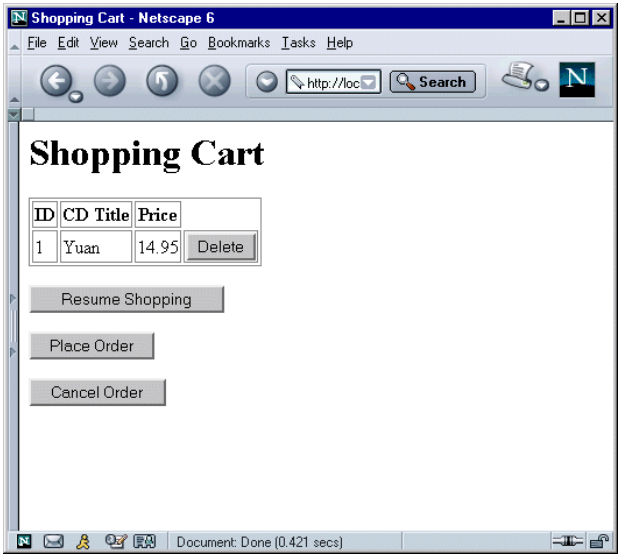
1. **The application starts with a CD Catalog List page that displays a list of CD titles.**

   This page is created with the ProductList JSP page.



2. **To add a CD to your shopping cart, click the Add button in the row of that CD.**

   This action displays the Shopping Cart page with the selected CD on it. This page is created by the ShopCart JSP page.



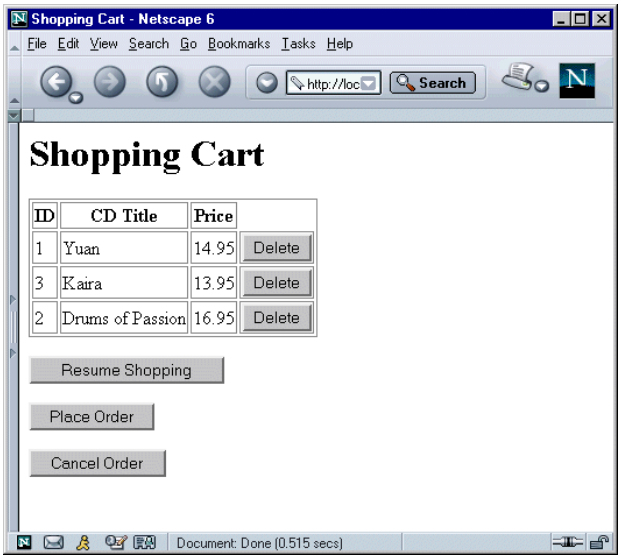
3. **To add another CD, click Resume Shopping, which takes you back to the CD Catalog List page.**

4. **Click Add on the same or a different CD.**

   The Shopping Cart page is redisplayed with an additional selection.

5. **Repeat Step 2 and Step 3 until you have all the CDs you want to purchase.**

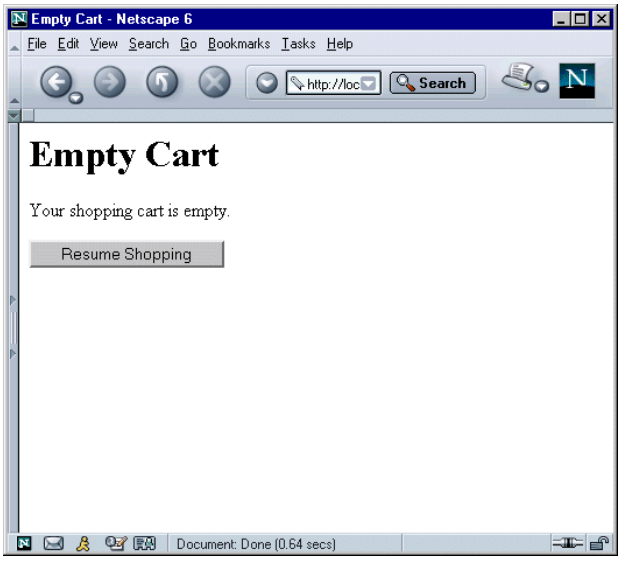
   The Shopping Cart displays your items. If you have chosen more than one of the same item, these are displayed as separate rows.



6. **To remove an item, click the item's Delete button.**

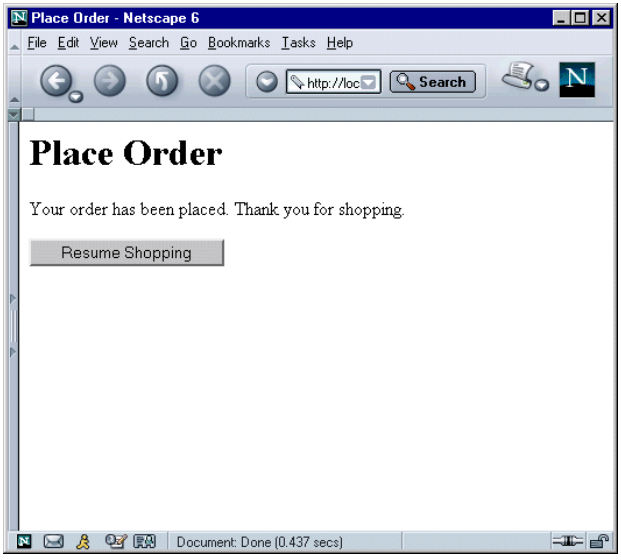
   The table is redisplayed minus the removed item.

   If you remove the last item in the table, the Empty Cart JSP page is displayed:



7. **Click the Resume Shopping button to return to the CD Catalog List page.**
8. **To place an order, click the Place Order button on the Shopping Cart page.**

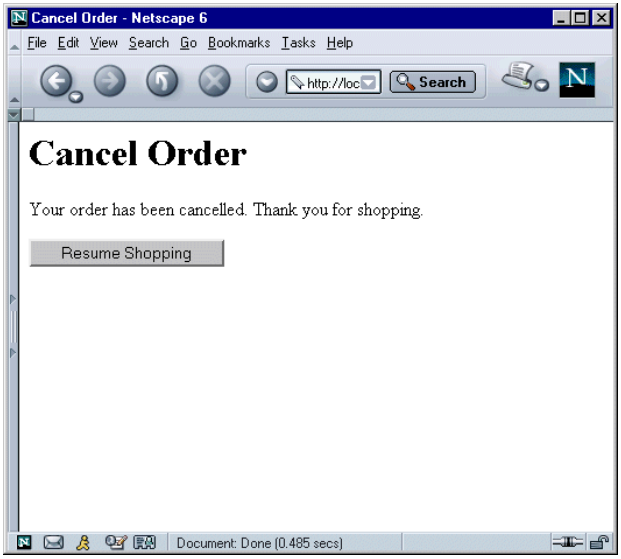
   The Place Order page is displayed. This page is created by the PlaceOrder JSP page.



You can either exit the application by pointing the browser at a different URL, or start a new session by clicking the Resume Shopping button.

9. **Or, to cancel your order, from the Shopping Cart page, click the Cancel Order button.**

   The Cancel Order page is displayed. This page is created by the CancelOrder JSP page.



   You can start a new session by clicking the Resume Shopping button.

Architecture of the Tutorial Application

The CDShopCart application is a web-centric application that uses a web client to send requests to and receive results from a web application. A *web application* is a bundle of web components and their supporting classes, beans, and files. *Web components* are server-side J2EE components, such as Servlets and JSP pages.

The CDShopCart application consists of a single web module. A *web module* is the smallest deployable and usable unit of web resources in a J2EE application. A feature introduced in the *Java Servlet Specification Version 2.3* and implemented by the Sun ONE Studio 4, Community Edition is the web module construct, which automatically creates the required directory structures, default versions of required data objects, and other special services required by the web module.

For more information on web modules and related concepts, see *Building Web Components*. For information specific to the web module construct, see the Developing Web Modules section under JavaServlet Pages and Servlets in online help.

FIGURE 2-1 shows the CDShopCart application elements and their relation to one another.

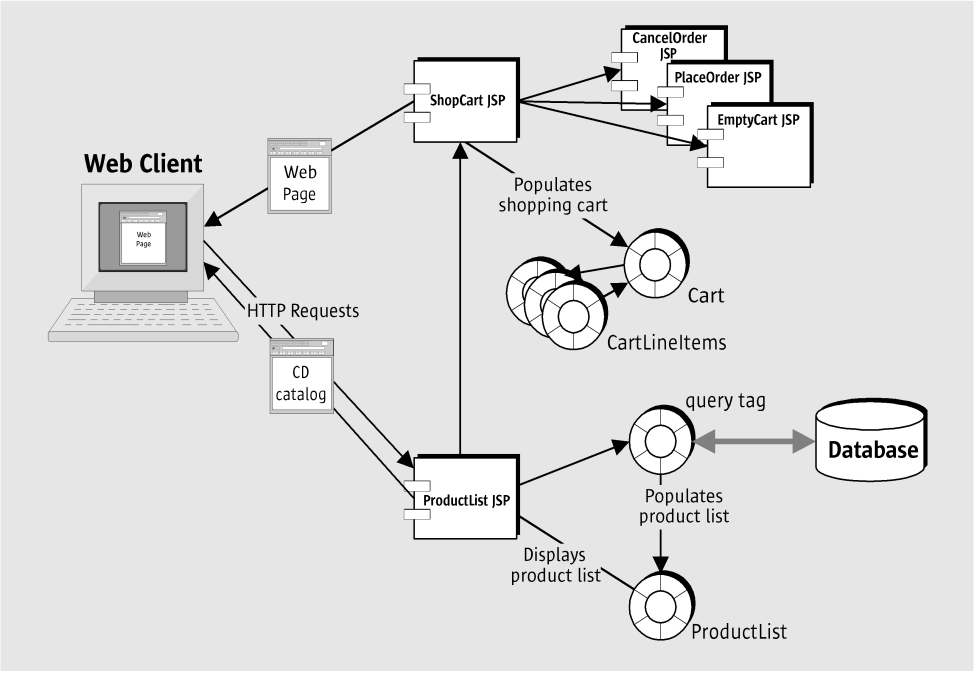


FIGURE 2-1 Architecture of the CDShopCart Application

Application Elements

Briefly, the elements shown in FIGURE 2-1 are:

- The *client* component

  The client component is a web browser that displays the application pages.

- The *service* component (a web module) that includes:
  - A `ProductList` JSP page that retrieves the product data from a database and displays it in a table on the CD Catalog List web page. ProductList also provides an Add button by which a user can add a CD to the shopping cart.
  - A `ShopCart` JSP page that displays CDs selected for purchase in a table on the Shopping Cart web page, and provides Delete, Place Order, Cancel Order, and Resume Shopping buttons.
  - An `EmptyCart` JSP page that displays a message when the customer deletes the last item in the shopping cart. Includes a Resume Shopping button.

- A `CancelOrder` JSP page that displays a message that the order has been canceled and a Resume Shopping button for returning to the ProductList JSP page.
- A `PlaceOrder` JSP page that displays a message that the order has been placed and a Resume Shopping button for returning to the ProductList JSP page.
- A `Cart` bean that represents the items selected for purchase.
- A `CartLineItem` bean that represents a cart line item.

Service Component Details

The service component of the CDShopCart application is a web module that includes four JSP pages that coordinate the application's behavior, given input from the client. Supporting elements to the web module include JavaBeans elements and an HTML page file.

- `ProductList` JSP page

  This page locates the session for the current user or creates one if it doesn't exist. `ProductList` uses SQL tags from the Apache Foundation's JSP Standard Tag Library (JSTL) to access the list of CDs from the database and core tags to display them in a table. `ProductList` also provides an Add button on each CD line item it displays.

- `ShopCart` JSP page

  When the user clicks the Add button on the `ProductList` page, the data of the line item is passed to this JSP page, which instantiates a `Cart` object consisting of `CartLineItem` objects, then uses core tags from the JSTL tag library to display them in a table. `ShopCart` provides a Delete button on each cart item. When this button is clicked, the page uses a scriptlet to remove the item, update the table data, and redisplay it. If the item removed is the last item in the cart, `ShopCart` forwards to the `EmptyCart` page. `ShopCart` also provides Resume Shopping, Cancel Order, and Place Order buttons. These buttons forward to the `ProductList` page, the `CancelOrder` page, and the `PlaceOrder` page, respectively.

- `Cart` JavaBeans component

  This bean has a lineItems attribute and includes the methods for getting and removing the `CartLineItem` objects. This bean is imported by the `ShopCart` page.

- `CartLineItem` JavaBeans component

  This bean has CD-related attributes, and includes methods for getting and setting attributes of Cart line items (ID, title, artist, country, and price).

- `CancelOrder` JSP page

  This JSP page is called when the user clicks the Cancel Order button on the `ShopCart` page. This page invalidates the session, displays a message that the order is cancelled, and provides a Resume Shopping button to return to the `ProductList` page.

- `EmptyCart` JSP page

  This JSP page is called when the user removes the last item from the `ShopCart` page. `EmptyCart` displays a message that the cart is empty and includes a Resume Shopping button.

- `PlaceOrder` JSP page

  This JSP page is called when the user clicks the Place Order button on the `ShopCart` page when there are items in the cart. `PlaceOrder` displays a message that the order has been placed, invalidates the session, and includes a Resume Shopping button.

Overview of Tasks for Creating the Tutorial Application

The tutorial consists of one chapter, in which you create the basic application. Before you can create the tutorial application, you must have the Sun ONE Studio 4 software installed and set up to run, and the tutorial database tables installed, as described in Chapter 1.

In Chapter 3 you learn how to use the following Sun ONE Studio 4, Community Edition features:

- Web modules (creating, developing, and test running)
- Embedded JSTL tags for connecting to and interacting with a database
- Embedded JSTL tags for iterating through and presenting the retrieved data

In addition, you create the supporting elements: several beans and an HTML page.

Creating a Web Module

The Sun ONE Studio 4 IDE provides tools for automatically creating the hierarchical directory structure of a web application. This structure is the web module. You develop the entire CDShopCart application within a single web module construct.

The tutorial doesn't try to provide complete information about how to develop a web module. The section "Creating a Web Module" on page 41 serves as an introduction to the subject, outlining the basic elements of the structure, and the (very easy) method for creating it. When you want to know more about how to develop web modules, see *Building Web Components* and online help.

Using the JSTL Tag Libraries

Within the CDShopCart web module, you create the ProductList JSP page to fetch the CD catalog data to display on the CD Product List web page. You then create the ShopCart JSP page to display CDs that the user selects for purchase. You use JSTL tags for database access and data presentation functions to accomplish this.

The section "Using JSP Tags to Fetch and Display Database Data" on page 44 describes how to use the SQL tags to make the JDBC connection to the database and fetch the CD data. It then describes how to use core tags to iterate through the resulting data, so that ProductList can display it in an HTML form on the web page.

The ShopCart JSP page displays CD data passed to it by the ProductList JSP page. The section "Adding Code to Add or Remove an Item From the Shopping Cart Table" on page 67 demonstrates how to use core JSTL tags to iterate through the passed values to find the individual field values, so they can be displayed in the correct columns in the cart table.

Creating the Supporting Elements

The supporting elements for the ShopCart JSP page are two beans (Cart and CartLineItem) and three JSP pages (CancelOrder, PlaceOrder, and EmptyCart).

The section "Creating the `CartLineItem` Bean" on page 57 shows you how to create a bean whose object holds the parameters of a line item passed to ShopCart from ProductList when a user clicks the Add button on the item. Then, in "Creating the `Cart` Bean" on page 63, you learn how to create a bean whose object holds the accumulated line items that have been selected. The Cart bean has methods for adding and removing line items from the cart.

In "Creating the Empty Cart Page" on page 72, you create a JSP page that displays a message that the cart is empty. This is to avoid displaying an empty form on the Shopping Cart page.

You create two more JSP pages that hold minor logic compared with ProductList and ShopCart. In “Creating the Place Order Page” on page 73, you create a JSP page that thanks the user for placing an order and then invalidates the session. In “Creating the Cancel Order Page” on page 75, you create a similar page that announces that an order is canceled and invalidates the session.

Testing the Application

Throughout this chapter, you test each element just after you create it. The IDE automatically deploys a web module to its internal container when you execute one of the web module’s components.

End Comments

The tutorial application is designed to be brief enough for you to create in a relatively short time (a day or so). This places certain restrictions on its scope. For example:

- There is no error handling.
- There are no debugging procedures.
- There is no description of how to create the WAR file for deployment.

Future releases will have these procedures.

Although the tutorial application is designed to be a simple application that you can complete quickly, you might want to import the entire application, view the source files, or copy and paste method code into methods you create. The CDShopCart application is accessible from the Sun ONE Studio 4 Developer Resources portal at `http://forte.sun.com/ffj/documentation/tutorialsandexamples.html`

CHAPTER 3

Creating the CDShopCart Application

This chapter describes, step by step, how to create the CDShopCart application. Before you can create the tutorial application, you must have the Sun ONE Studio 4 software installed and set up to run, and you must have the tutorial database table installed, as described in Chapter 1.

This chapter is organized under the following topics:

You test each component as you create it. By the end of this chapter, you will be able to run the basic application, as described in Chapter 2.

Tip – Complete source code for all JSP pages and JavaBeans components described in this chapter is found in Appendix A.

Creating a Web Module

The CDShopCart application is a web application. Web applications consist of web modules. The CDShopCart application is a very simple application, containing only one web module.

This section shows you how to implement the shopping cart features within a web module by using the Sun ONE Studio 4, Community Edition IDE.

What Is a Sun ONE Studio 4 Web Module?

According to the *Java Servlet Specification, version 2.3*, "a web application exists as a structured hierarchy of directories." The root of this hierarchy is the document root, which holds all the files that are part of the web application. The hierarchy also includes a special non public subdirectory, the `WEB-INF` directory, for items that are related to the web application but are not to be served directly to the client. Items in the `WEB-INF` directory include the web deployment descriptor (the `web.xml` file) and servlet and utility classes used by the web application loader for loading classes.

Your application's files must eventually be part of a web module structure to package them as a WAR file (a Web ARchive format file) for delivery into a web container. The Sun ONE Studio 4 IDE's web module feature automates much of the process of creating the required directory hierarchy, as well as filling in the hierarchy with default versions of some of the objects.

Note – This tutorial does not try to provide complete information about developing web modules. For that task, see the *Building Web Components* book. Also, consult the Sun ONE Studio 4 online help for more details on web modules.

Creating the `CDShopCart` Web Module

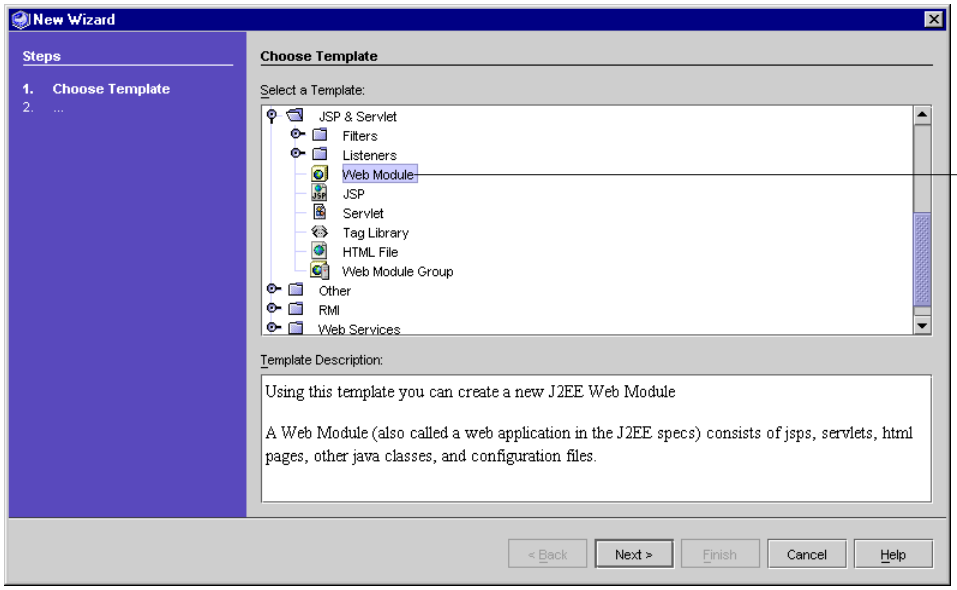
In this section, you create the web module for the CDShopCart application. You'll use the Sun ONE Studio 4 web module feature to build this web module directory from scratch, although you could also convert an existing directory into a web module.

To create the `CDShopCart` web module:

1. **In the Filesystems pane of the Explorer window, choose File → New to display the New wizard.**

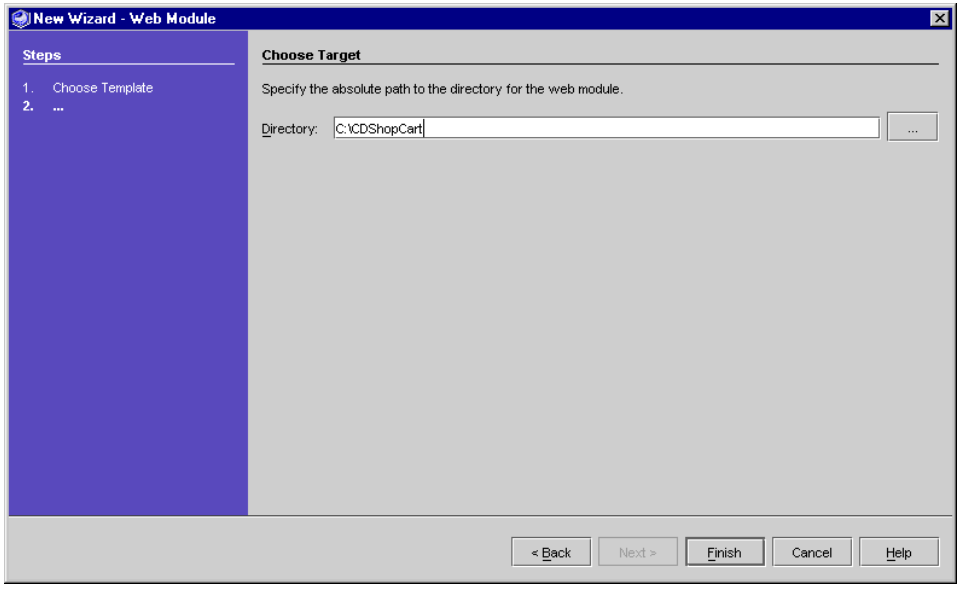
   You can use this wizard to create many different types of objects from the provided templates.

2. **Open the `JSP & Servlet` node and select `Web Module`.**



3. **Click Next to display a pane for specifying the web module directory name.**

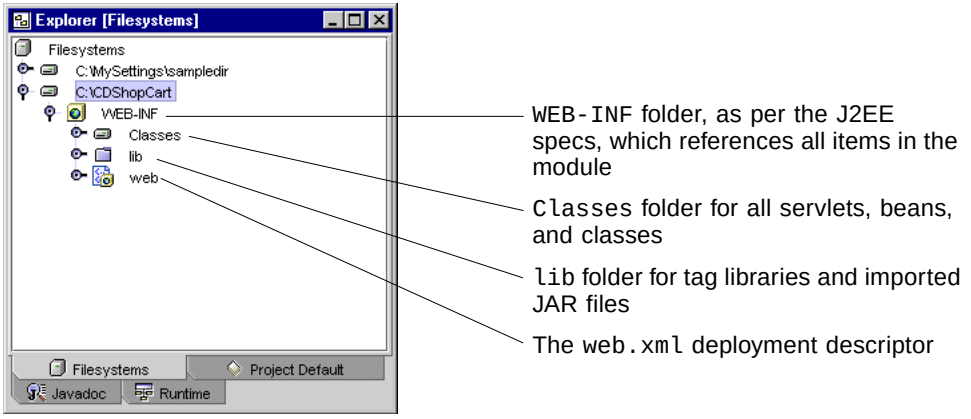
4. **In the Directory field, erase the default directory filespec and type `c:\CDShopCart`.**



5. **Click Finish.**

   The new `CDShopCart` web module is created in the Explorer. A dialog box appears, stating that an alternate view of the web module is installed in the Default Project window. Click OK to dismiss this message.

6. **Open the nodes in the web module to reveal what has been created automatically.**



Now you are ready to create the first component of the application, the `ProductList` JSP page.

Using JSP Tags to Fetch and Display Database Data

In this section, you create a JSP page named `ProductList` that fetches and displays the CD product data. To connect to the database, retrieve the table data, and format the data for display, you'll use JSP Standard Tag Library (JSTL) tags. JSTL is from the Jakarta Project, a project of the Apache Foundation. JSTL is embedded in Sun ONE Studio 4.

For complete information on JSTL, including descriptions, examples, and a tutorial, see the Jakarta Project web site at `http://jakarta.apache.org/taglibs/index.html`.

What Is a JSP Tag?

The body of a JSP file can contain two types of code: fixed template data and elements.

- Fixed template data—code type not known to the JSP container; this data passes through the HTTP response unchanged.

  Examples of fixed template data are XML and HTML code. You'll use HTML code in the CDShopCart application to create headings, titles, tables, and buttons.

- Element types—include three different types:
  - Directives—used to declare global information about the JSP page, such as which packages to import and whether the JSP page must join a session.
  - Scripting elements—enable you to embed Java code within a JSP file.
  - JSP tags—XML-style elements that provide a way to work with Java objects without having to write Java code.

    Java classes associated with each tag implement the tag's functionality.

Standard Tags and Custom Tags

Standard tags, which are defined in the JSP specification document, are available in any JSP container. Custom tags are tags defined in an XML document called a *tag library*. A custom tag library is made available to the JSP page by means of a declaration in a directive element.

JSTL Tags

The JSP Standard Tag Library is a custom tag library produced by the Jakarta Project. Embedded in the Sun ONE Studio 4 IDE are JAR files that contains the JSTL tags and supporting classes and interfaces. This files are:

- `standard.jar`—contains multiple JSTL tag libraries, including core and SQL tags
- `jstl.jar`—contains supporting interfaces and classes for the JSTL libraries in `standard.jar` used by CDShopCart

The two tag library descriptor (TLD) files you will use in these JAR files are `sql.tld` (for database actions) and `core.tld` (for everything else).

Using JSTL Tags

To use any JSP tags, you need to declare the tag library, and then follow the prescribed syntax of the particular tag.

For complete information on JSTL syntax conventions, see the JSTL documentation, available from the Jakarta Project web site at `http://jakarta.apache.org/taglibs/index.html`.

Using the `taglib` Directive

To use tags in a JSP page, you must first declare the tag library with a `taglib` directive.

The `taglib` directive must declare that the page uses the tag library of a given URI (Uniform Resource Identifier) and specify the tag prefix that is used in calls to actions in the library. The URI and the prefix for JSTL tags are defined with Apache JSTL conventions.

The generic syntax for the directive is:

```
<%@ taglib prefix="prefix" uri="http://java.sun.com/jstl/taglibname
```

For example, to declare the `core` taglib:

```
<%@ taglib prefix="c" uri="http://java.sun.com/jstl/core
```

Using Tag Syntax

JSP tags are based on XML syntax and have one of two forms:

- Start tag (the element name) plus possible attribute/attribute value pairs, an optional body, and a matching end tag
- Empty tag with possible attributes

An example start tag you'll use in CDShopCart is the `query` tag, which fetches data from the data source. The `query` tag uses the following syntax:

```
<sql:query var="stored_query" dataSource="${dataSource}" >
        body
</sql:query>
```

An example empty tag you'll use is the `sql` library `setDataSource` tag, which creates a JDBC connection to a database. The `setDataSource` tag uses the following syntax:

```
<sql:setDataSource var="dataSource"
       url="driver_url"
       driver="driver_string"
       user="user_id" password="pwd"/>
```

A JSTL tag convention is to use "var" for any tag attribute that exports information about the tag. Note that "var" was chosen to emphasize the fact that this is a JSP variable, not a scripting variable, which uses the convention "id" for a variable.

Creating the CD Catalog List Page

This section describes how to create the mechanism for retrieving data from the database you installed in Chapter 1 and displaying it in a table for the user. The page you create looks like FIGURE 3-1.

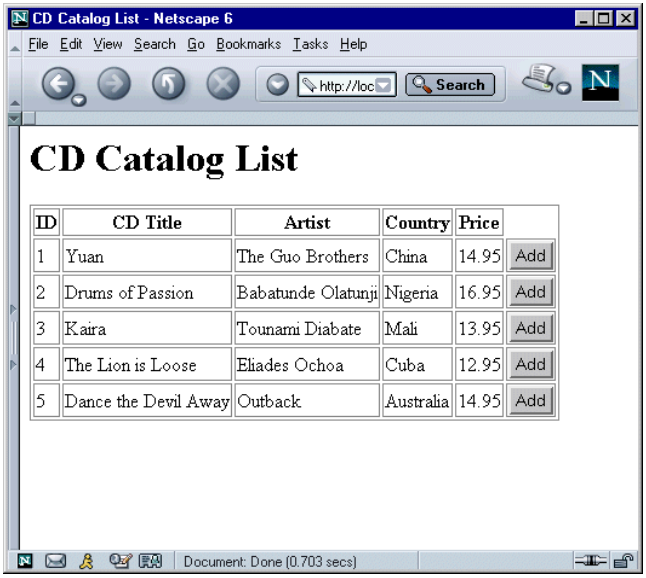


FIGURE 3-1 CD Catalog List Page

Creating this page includes the following tasks:

1. “Adding JSTL Tag Libraries to the Web Module” on page 48
2. “Creating the `ProductList` JSP Page” on page 49
3. “Declaring the Tag Libraries” on page 50
4. “Using the `setDataSource` Tag to Connect to the Database” on page 51
5. “Using the `query` Tag to Fetch the CD Data” on page 52
6. “Using the Iteration Tags to Display the Data” on page 52
7. “Creating the Add Button for Each CD Row” on page 53

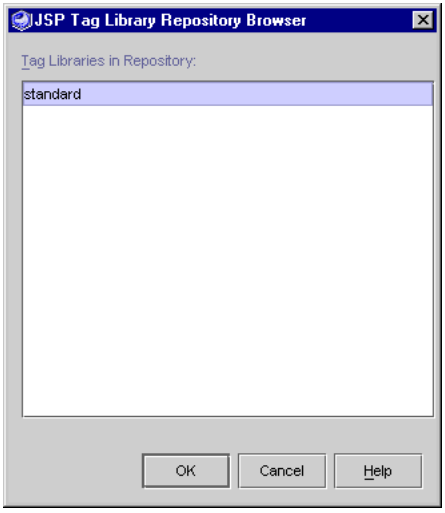
Adding JSTL Tag Libraries to the Web Module

In this section, you import the JSTL tag library JAR file, `standard.jar`, and the JAR file of supporting classes and interfaces, `jstl.jar`, to the `CDShopCart` web module, because you will use actions that are implemented by the these files.

To import JSTL tag libraries into the web module:

1. **In the Explorer, select the `CDShopCart` web module and choose Tools → Add JSP Tag Library → Find in Tag Library Repository.**

   The JSP Tag Library Repository Browser appears.

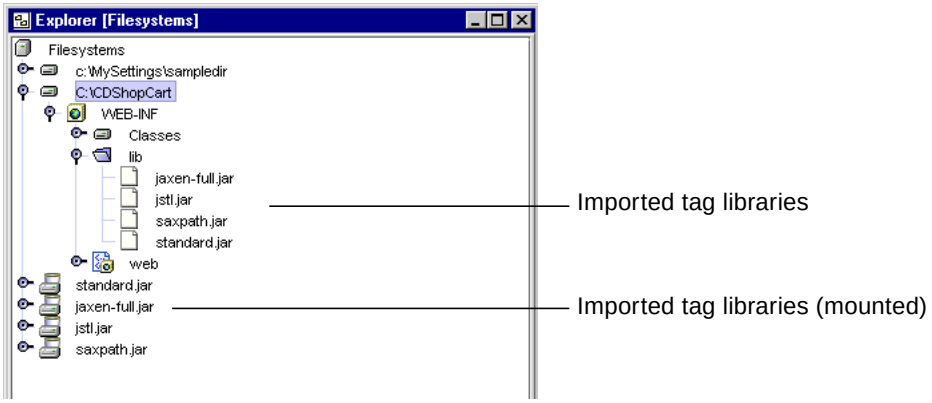


2. **Select the `standard` file and click OK.**

   In the Explorer, expand the `lib` node under the `WEB-INF` node.

3. **Check the files are under the `lib` node.**

   The `jstl.jar` and `standard.jar` files, which contain the tags you use in this tutorial, are displayed under the `lib` node. Two additional jar files are also displayed. These contain APIs required for XML tags, which you will not use. All four files are also separately mounted. The Explorer looks like this:



Tip – Right-click the top-level `Filesystems` node and choose Customize. The window that opens displays all the files that are now mounted in your Sun ONE Studio 4 CLASSPATH. You should be able to see the two JAR files in the list.

Creating the `ProductList` JSP Page

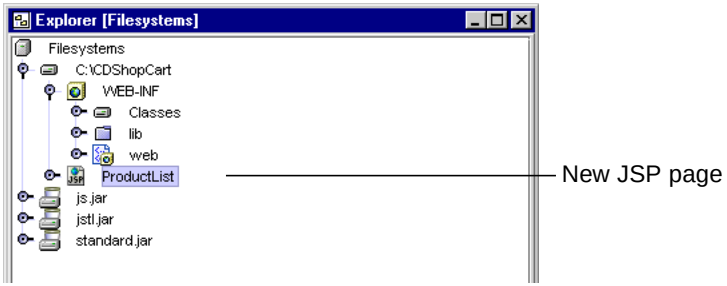
Now, create the page that uses the tags to retrieve the CD data from the database and display the data in a table. The title of this page is CD Catalog List, and the mechanism that produces it is the `ProductList` JSP page.

1. **In the Explorer, right-click the `CDShopCart` web module and choose New → JSP & Servlet → JSP from the contextual menu.**

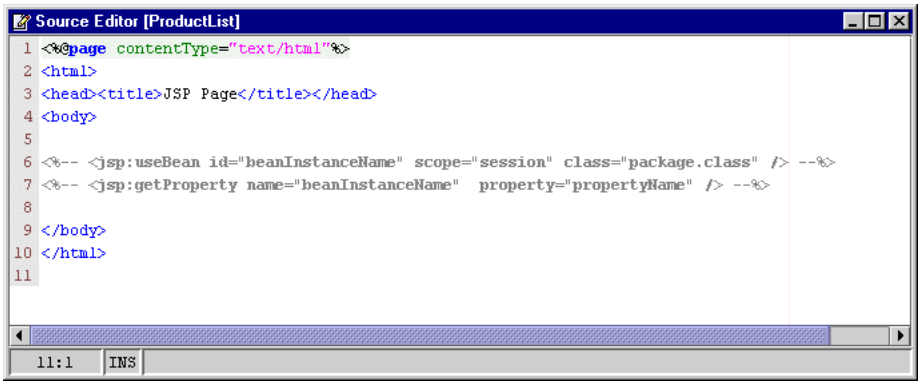
   The New wizard appears displaying the New Object Name pane.

2. **Type `ProductList` in the Name field and click Finish.**

   The `ProductList` JSP page is displayed in the web module.



A JSP page skeleton is displayed in the Source Editor.



Declaring the Tag Libraries

You must first declare the tag libraries as described in "Using the `taglib` Directive" on page 46. Put the directive above the body of the JSP page, right under the page title. The following procedure shows how to change the title of the page and add the directives for the two tag libraries.

1. **In the body of the `ProductList` page, change the document title to CD Catalog List:**

```
<head><title>CD Catalog List</title></head>
```

2. **Add directives to import the core and SQL actions tag libraries, as follows:**

```
<%@taglib prefix="c" uri="http://java.sun.com/jstl/core" %>
<%@taglib prefix="sql" uri="http://java.sun.com/jstl/sql" %>
```

Using the `setDataSource` Tag to Connect to the Database

The first tag you'll use is the `sql` library `setDataSource` tag, which creates a JDBC connection to a database. First add a page heading (with an `<H1>` tag), then add the `setDataSource` tag below the `<body>` HTML tag to connect to the database.

1. **Below the `<body>` tag in the ProductList page, create a page title:**

```
<body>
<h1> CD Catalog List </h1>
```

2. **Below the header, create the JDBC connection.**
   - The following is for a PointBase driver:

```
<sql:setDataSource var="productDS"
      url="jdbc:pointbase:server://localhost/cdshopcart"
      driver="com.pointbase.jdbc.jdbcUniversalDriver"
      user="PBPUBLIC" password="PBPUBLIC" />
```

   - If you're using an Oracle database (using the thin driver), use this:

```
<sql:setDataSource var="productDS"
      url="jdbc:oracle:thin:@hostname:port#:SID"
      driver="oracle.jdbc.driver.OracleDriver"
      user="userid" password="password" />
```

     The default Oracle port number is 1521.

   - If you're using a Microsoft SQLServer database with a Weblogic driver, use this:

```
<sql:setDataSource var="productDS"
      url="jdbc:weblogic:mssqlserver4:database@hostname:port#"
      driver="weblogic.jdbc.mssqlserver4.Driver"
      user="userid" password="password" />
```

     The default port number for SQLServer is 1433.

Tip – To automatically reformat code after you have entered it, put the cursor in the Source Editor and press Ctrl-Shift-F.

Using the query Tag to Fetch the CD Data

Now use the sql tag, query to query the database and get a single result set containing rows of data and store it in the productQuery result set. You'll then pass this JSP variable to iterator tags to display the results. The query tag supports the standard SQL statement SELECT.

Put the query tag just after the driver tag.

To query the database for all the CD data:

- **In the** ProductList **page, just after the** driver **tag, create a query to select all the CD data from the database (use the** productDS **dataSource created above):**

```
<sql:query var="productQuery" dataSource="${productDS}" >
       SELECT * FROM CD
</sql:query>
```

Using the Iteration Tags to Display the Data

You need to create a table and fill the table cells with the data. First, create a table with HTML tags, then use the forEach tag to iterate through the data you just fetched. Use the out tag to fetch the field data of each row.

The syntax for forEach is:

```
<c:forEach var="$Current_collection_item" items="${Collection}" >
```

The items variable holds the current collection that is the target of the iteration, and var holds the current item of that collection. If the collection is a result set, then the current item is the result set object positioned at the current row. For example, to iterate over a row from a result named myResultSet:

```
<c:forEach var="row" items="${myResultSet.rows}" >
       <TR>
           <TD><c:out value="${row.col1}"/></TD>>
           <TD><c:out value="${row.col2}" /></TD>
           <TD><c:out value="${row.col3}" /></TD>
       </TR>
</c:forEach>
```

To create the table and fill the cells with data:

1. **Start a table for the CD data:**

```
<TABLE border=1>
       <TR>
           <TH>ID</TH>
           <TH>CD Title</TH>
           <TH>Artist</TH>
           <TH>Country</TH>
           <TH>Price</TH>
       </TR>
```

2. **Next, use the** `forEach` **and** `out` **tags to populate the table:**

```
<c:forEach var="row" items="${productQuery.rows}" >
       <TR>
           <TD><c:out value="${row.ID}"/></TD>
           <TD><c:out value="${row.CDTITLE}"/></TD>
           <TD><c:out value="${row.ARTIST}"/></TD>
           <TD><c:out value="${row.COUNTRY}"/></TD>
           <TD><c:out value="${row.PRICE}"/></TD>
```

You will add more to this code in the next section.

Creating the Add Button for Each CD Row

Each row of the CD table holds the data for one CD. To purchase a CD, the user clicks the Add button on each row. Create an HTML form to define the area for user input (clicking the button) and within this area, embed information that will be passed to the Shopping Cart JSP page. Put this code immediately after the previous code.

To create the Add button:

1. **Create a form for the table within a cell:**

```
<TD>
<form method=get action="ShopCart.jsp">
```

2. **Specify the embedded information for product ID, title, and price:**

```
      <input type=hidden name=cdId value="<c:out value=
"${row.ID}"/>">
      <input type=hidden name=cdTitle value="<c:out value=
"${row.CDTITLE}"/>">
      <input type=hidden name=cdPrice value="<c:out value=
"${row.PRICE}"/>">
```

3. **Directly underneath the preceding code, specify the Add button:**

```
<input type=submit name=operation value=Add>
```

4. **End the form, the cell, and the row:**

```
</form>
</TD>
      </TR>
```

5. **End the iteration and the table:**

```
</c:forEach>
</TABLE>
```

6. **Choose File → Save to save your work.**

Tip – To see the completed source code of this page, see "`ProductList.jsp` Source" on page 80.

Testing the `ProductList` JSP Page

Now validate your work. Compile the `ProductList` page and use the integrated Sun ONE Studio 4 runtime system and your web browser to execute the page.

To test the `ProductList` page:

1. **If not already started, start the PointBase Network Server by choosing Tools → PointBase Network Server → Start Server.**

   A PointBase server window is displayed.

2. **Make sure Tomcat is the default web server.**

   Select the Runtime tab of the Explorer. Expand the Server Registry node, then the Default Servers node. If Tomcat4.0 is not designated for Web Tier Applications, expand the Installed Servers node and all subnodes under the Tomcat 4.0 node. Right-click `localhost:8081` and choose Set As Default.

3. **In the Filesystems pane of the Explorer, select the `CDShopCart` web module and choose Build → Build All.**

   Watch the message area in the lower part of the toolbar for status messages. If all is well, you see "Finished." If there are problems, the output window displays an error message, with the problem line. Fix any problems and redo this step until you see the "Finished" message.

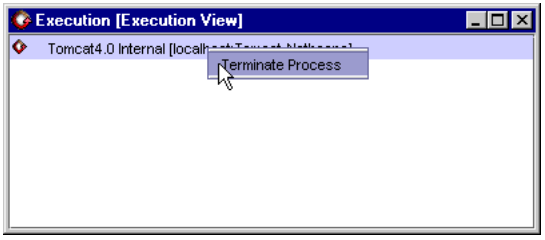
4. **Execute the `ProductList` JSP page by selecting it and clicking the Execute button in the toolbar ( ).**

   Alternatively, choose Build → Execute, or right-click `ProductList` and choose Execute from the contextual menu.

   The IDE starts the built-in Tomcat server, switches to the running workspace, and opens the Execution window. When the Servlet is running, a message is displayed in the Execution window and the browser opens. After a few seconds, the CD Catalog List page is displayed, as in FIGURE 3-1.

5. **Terminate the execution by right-clicking the process in the Execution window and choosing Terminate Process.**

   The process you are terminating is the Tomcat Web Server process.



6. **Return to the editing workspace by clicking the Editing tab.**

   Congratulations! You have successfully created a JSP page that uses JSTL tags to open a connection to a database and retrieve and display data from it. Now create the Shopping Cart page.

Creating the Shopping Cart Page and Supporting Elements

Now create the mechanism for displaying items selected for purchase from the CD Catalog List page. This mechanism includes:

- A bean (`CartLineItem`) to hold the attributes of the selected CD row passed as parameters from the `ProductList` page
- A beaninfo component (`CartLineItemBeanInfo`) to specify that `id` and `price` are properties of `CartLineItem`, despite having overloaded setter methods
- Another bean (`Cart`) to hold the `CartLineItem` objects
- The `ShopCart` JSP page to receive the `Cart` objects and display them as a row in a table
- A Delete button for each item displayed on the `Shopping Cart` page
- Cancel Order, Resume Shopping, and Place Order buttons with their implementations

When a few items have been selected, the `Shopping Cart` page you create looks like FIGURE 3-2.

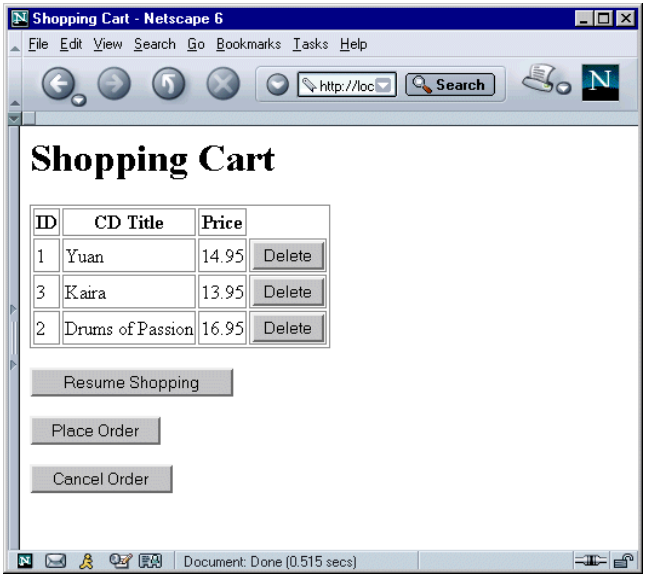


FIGURE 3-2 Shopping Cart Page

Use the following procedures to create the ShopCart JSP page and its beans:

Creating the CartLineItem Bean

Create a line item bean whose object can hold the parameters passed to the Shopping Cart page from the CD Catalog List (ProductList) page. To do this, create three properties on the bean with their accessor methods.

Note – J2SE 1.4 requires all classes to be contained within packages, and this tutorial is compliant with J2SE 1.4. Since the classes subdirectory of the WEB-INF directory is not a package, you must create a package under the classes subdirectory to hold your classes (beans). This package is called the ShopCart package in this tutorial. This change also works if you are using J2SE 1.3.1.

---

1. **Expand the** WEB-INF **node of the** CDShopCart **web module, right-click the** Classes **node, and choose New → Java Package.**
2. **Name the package** ShopCart**.**
3. **Right-click the** ShopCart **package and choose New → Beans → Java Bean.**

   The New wizard appears displaying the New Object Name pane.
4. **In the Name field, type** CartLineItem **and click Finish.**

   The new CartLineItem bean is displayed in the Explorer, and its code in the Source Editor. The red badge near the bean ( ) is a "need to compile" indicator. Don't worry about this; you will compile later.
5. **Expand the bean and its class to reveal its contents.**
6. **Right-click the** Bean Patterns **node and choose Add → Property.**
7. **In the New Property Pattern dialog box, define the** cdtitle **property.**

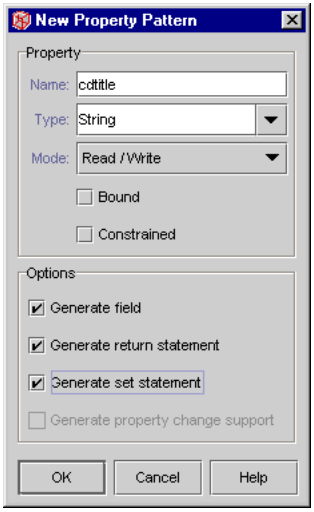
   a. **Type cdtitle in the Name field.**

   b. **Select** String **for the Type.**

c. Select the following options:

- Generate field
- Generate return statement
- Generate set statement

The dialog box should look like this:



8. **Click OK to accept the information and close the dialog box.**
9. **Repeat Step 6 through Step 8 to create the `id` property in the same way.**

   a. Type `id` in the Name field.

   b. Select `int` for the Type.

   c. Select the following options:

   - Generate field
   - Generate return statement
   - Generate set statement

10. **Repeat Step 6 through Step 8 again to create the `price` property in the same way.**

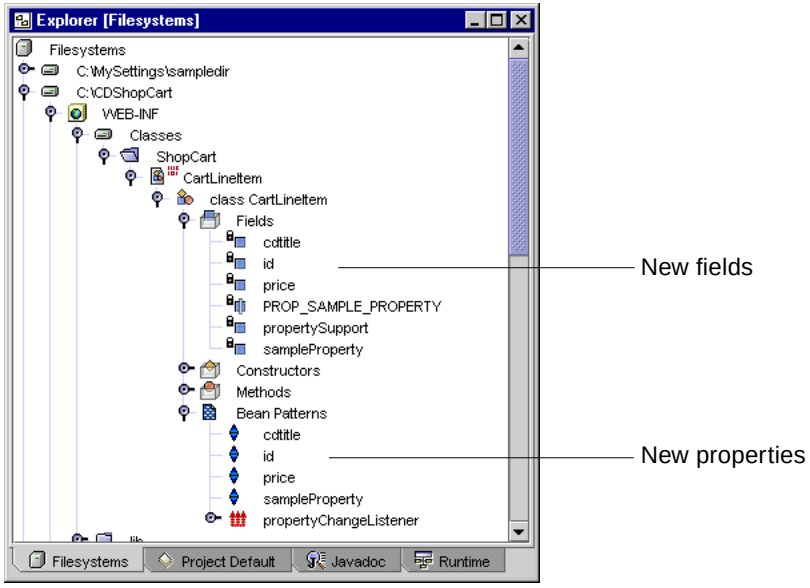
    a. Type `price` in the Name field.

    b. Select `double` for the Type.

    c. Select the following options:

    - Generate field
    - Generate return statement
    - Generate set statement

11. **Expand the `Fields` node (of the `CartLineItem` bean class) to see the new fields you created.**



12. **Expand the `Methods` node to see the new `get` and `set` methods for each field.**

13. **Double-click one of the new fields you just created (for example, `cdtitle`) to see the code for it in the Source Editor.**

Converting the `id` Property and `price` Property to Strings

The parameters passed from the `ProductList` JSP page are all passed as strings. However, because neither the `id` property nor `price` property are strings, you must convert them. An efficient way to do this is to overload the properties' setter methods and add the proper code.

To overload the `setId` method and `setPrice` methods:

1. **Right-click the `Methods` node (of the `CartLineItem` bean), and choose Add Method.**

2. **In the Edit New Method dialog box that appears, define the setId method.**

   a. **Type `setId` in the Name field.**

   b. **Select `void` for Return Type.**

3. **In the Method Parameters box, click the Add button to display the Enter Method Parameter dialog box.**

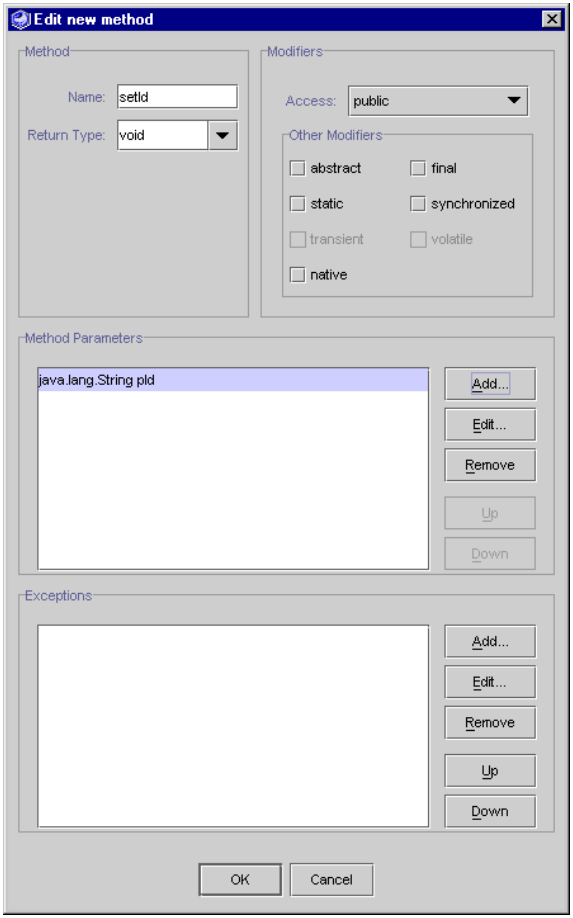
4. **Define the `pId` parameter.**

   a. **Select `java.lang.String` for the Type.**

   b. **Type `pId` for the Name.**

5. **Click OK.**

   The New Method dialog box looks like this:



6. **Click OK to create the method and close the dialog box.**

7. **Add code to this new method, as follows:**

```
public void setId(java.lang.String pId) {
       int val = Integer.parseInt(pId);
       this.setId(val);
}
```

Tip – Whenever you enter code into the Source Editor by copying and pasting, you can automatically reformat it properly by putting the cursor in the Source Editor and typing Ctrl-Shift-F.

Now do the same with the `setPrice` method.

8. **Repeat Step 1 through Step 6 to define the `setPrice` method.**

   For the method definition:

   a. **Type `setPrice` in the Name field.**

   b. **Select `void` for Return Type.**

   For the parameter definition:

   a. **Select `java.lang.String` for the Type.**

   b. **Type `pPrice` for the Name.**

9. **Add the following code to this new method:**

```
public void setPrice(java.lang.String pPrice) {
       double val = Double.parseDouble(pPrice);
       this.setPrice(val);
}
```

10. **Select the `CartLineItem` bean (), not the class () and choose Build → Compile.**

    If the bean compiles without errors, the red "need to compile" badge is removed from the bean's node and you are ready to create the `Cart` bean. If not, check your typing and recompile.

Tip – The completed source code for this bean is found at "`CartLineItem` Bean Source" on page 81.

Creating the `CartLineItemBeanInfo` Component

It is a standard Java convention, built in to the Introspector class, that a bean property is a private attribute that is accessible by a pair of public accessor (get and set) methods. Because you have just overloaded the set methods for the `id` and `price` properties, the Introspector will no longer recognize these as bean properties.

However, you do want these fields to be recognized as properties, because the JSTL expression language will only make bean properties available if they follow Java conventions. For example, item that comes after the "." operator in the following code must be a property:

```
<TD><c:out value="${row.id}"/></TD>
<TD><c:out value="${row.cdtitle}"/></TD>
```

You can work around this difficulty with a BeanInfo component, with which you can specify the properties of a bean directly. Create a BeanInfo component for the `CartLineItem` bean, then code it to specify that `id` and `price` are properties.

To specify that `id` and `price` are properties of the `CartLineItem` bean:

1. **Right-click the ShopCart package and choose New → Beans → BeanInfo w/o Icon.**
2. **Name the BeanInfo `CartLineItemBeanInfo` and click Finish.**

   The new `CartLineItemBeanInfo` node is displayed in the Explorer.
3. **Expand the BeanInfo node and its `Methods` node.**
4. **Double-click the `getPdescriptor` method.**

   The `CartLineItemBeanInfo` bean code is displayed in the Source Editor at the definition of the `getPdescriptor` method.

5. **Add the following code to the body of the** `getPdescriptor` **method:**

```
if (properties == null) {
       try {
           PropertyDescriptor[] props = {
               new PropertyDescriptor("cdtitle",
CartLineItem.class),
               new PropertyDescriptor("id", CartLineItem.class),
               new PropertyDescriptor("price",
CartLineItem.class)};
               properties = props;
       } catch (IntrospectionException ex) {
           return null;
       }
}
```

Tip – Remember, whenever you enter code into the Source Editor by copying and pasting, you can automatically reformat it properly by putting the cursor in the Source Editor and typing Ctrl-Shift-F.

---

6. **Right-click the** `CartLineItemBeanInfo` **node and choose Compile.**

   The BeanInfo bean should compile without errors.

Creating the `Cart` Bean

The `ShopCart` JSP page instantiates (or finds, if it already exists) a `Cart` object to hold the CD line item objects that are passed to it by the `ProductList` JSP page whenever a user clicks the Add button. The `cart` object is based on a `Cart` bean.

To create the `Cart` bean:

1. **Right-click the** `ShopCart` **package and choose New → Beans → Java Bean.**
2. **Name the bean** `Cart` **and click Finish.**
3. **Right-click the** `Cart` **bean's** `Bean Patterns` **node and choose Add → Property.**
4. **With the New Property Pattern dialog box, create a new** `lineItems` **bean pattern.**

   a. **Type `lineItems` in the Name field.**

   b. **Type `java.util.Vector` in the Type field.**

c. **Select the following options:**

- Generate field
- Generate return statement
- Generate set statement

d. **Click OK to create the bean pattern and close the dialog box.**

5. **Double-click the new `lineItems` field (or the new accessor methods) to see the new code that was created in the Source Editor.**

   You must change the access of the field from private to public.

6. **Open the `Fields` node of the `Cart` class and select the `lineItems` field.**

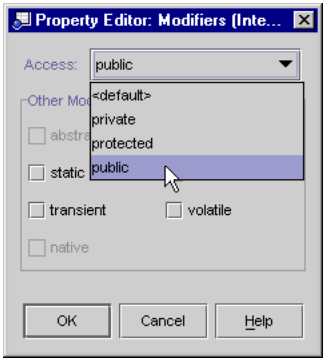
7. **Open the `lineItems` Properties window and click the Modifiers field.**

Note – If the Properties window is not already open (it is usually displayed under the Explorer window), right-click `lineItems` and choose Properties from the contextual menu. If you want the Properties window to remain open, choose Window → Windows → Browsing → Properties Window.

---

8. **Click the ellipsis (...) button to display the Modifiers dialog box.**

9. **Choose Public from the Access list.**



10. **Click OK to accept the change.**

    Now add code that instantiates a line item object, a method that returns the element number of a selected item, and another method that removes a line item from the cart.

11. **Open the Constructors node of the `Cart` bean and double-click the `Cart()` constructor.**

    This action takes you to the `Cart()` constructor in the Source Editor.

12. **Add the following (bold) code to the `Cart` bean's constructor to instantiate a new `lineItems` object, as follows:**

```
public Cart() {
       propertySupport = new PropertyChangeSupport ( this );
       lineItems = new java.util.Vector();
}
```

13. **Right-click the Cart `Methods` node, choose Add Method, and define the `findLineItem` method.**

    For the method definition:

    a. **Type `findLineItem` in the Name field.**

    b. **Select `int` for Return Type.**

    For the parameter definition:

    a. **Select `int` for the Type.**

    b. **Type `pID` for the Name.**

14. **In the Source Editor, add the following (bold) code to the `findLineItem` method:**

```
public int findLineItem(int pID) {
       System.out.println("Entering Cart.findLineItem()");
       //Return the element number of the item in the cartItems
       //as specified by the passed ID.
       int cartSize = (lineItems == null) ? 0 : lineItems.size();
       int i;
       for (i = 0; i < cartSize; i++ ) {
           if  ( pID ==
               ((CartLineItem)lineItems.elementAt(i)).getId() )
           break;
       }
       if (i >= cartSize) {
           System.out.println("Couldn't find line item for ID: " +
               pID);
           return -1;
       }
       else
           return i;
}
```

15. **Repeat Step 13, and define the `removeLineItem` method.**

    For the method definition:

    a. **Type `removeLineItem` in the Name field.**

    b. **Select `void` for Return Type.**

    For the parameter definition:

    a. **Select `int` for the Type.**

    b. **Type `pID` for the Name.**

16. **Add the following code in the Source Editor to the `removeLineItem` method:**

```
public void removeLineItem(int pID) {
       System.out.println("Entering cart.removeLineItem()");
       int i = findLineItem(pID);
       if (i != -1) lineItems.remove(i);
       System.out.println("Leaving cart.removeLineItem()");
}
```

17. **Select the `Cart` bean (), not the class (), and click the Compile button to compile the `Cart` bean.**

    The bean should compile without errors. Now create the `ShopCart` JSP page.

Tip – The completed source code of this bean is found at “`Cart` Bean Source” on page 87.

Creating the Shopping Cart Page

Now create the page that receives the parameters passed from the CD Catalog List page and displays some of them (id, title, and price) as a row in a table. This page also offers mechanisms for deleting an item from the table, returning to the Catalog List page, and placing the order. The title of this page is “Shopping Cart,” and the mechanism that produces it is the `ShopCart` JSP page.

1. **Create a JSP page by right-clicking the `CDShopCart` web module and choosing New → JSP & Servlet → JSP.**

2. **Name the JSP page `ShopCart` and click Finish.**

   The `ShopCart` JSP is displayed in the Explorer and in the Source Editor.

Develop this page by performing the following tasks:

1. “Adding Code to Add or Remove an Item From the Shopping Cart Table” on page 67.
2. “Using the Iteration Tags to Display the Data” on page 52.
3. “Adding the Buttons to the Page” on page 70.

Adding Code to Add or Remove an Item From the Shopping Cart Table

In this section, you add code that creates the cart items table. To instantiate a `Cart` object and a `CartLineItem` object, use a directive to import the `Cart` bean, the `java.util` library (the `CartLineItem` is a type Vector, from this library), and `CartLineItem`. Use the same core tags that you used in the `ProductList` JSP page, so also add a directive to import these tags.

Use scriptlet code to create the cart. Then add code that adds a line item created with the parameters passed from the `ProductList` JSP page. Add more code that deletes a line item when the user presses the Delete button. Add code to forward the action to a JSP page you will create later (the `EmptyCart` JSP page) if the table is empty.

As with the `ProductList` JSP page, use iteration tags to organize the table data. Then use a form to create the table and add a Delete button to each row.

1. **Add a `Page` directive to import the `java.util` library and the `ShopCart` package:**

```
<%@page contentType="text/html" %>
<%@page import="java.util.*, ShopCart.*" %>
```

2. **Change the page title to Shopping Cart and add the directive to import the `core.jar` library:**

```
<head><title>Shopping Cart</title></head>
<%@taglib prefix="c" uri="http://java.sun.com/jstl/core" %>
```

3. **Below the `<body>` tag, create a Shopping Cart heading for the page:**

```
<body>
<h1> Shopping Cart </h1>
```

4. **Below the heading, use the** `usebean` **tag to tell the JSP page to use the** `Cart` **bean:**

```
<jsp:useBean id="myCart" scope="session" class="ShopCart.Cart" />
```

This code instantiates a `Cart` object and places it on the session.

Now specify what happens when the current operation for the session is Add. This happens when the user clicks the Add button on the `ProductList` page. Add code that gets the `cdID`, `cdTitle`, and `cdPrice` objects and adds them to the `myCart` object.

5. **Begin the code by getting the current operation and defining what happens when a user clicks Add:**

```
<%
       String myOperation = request.getParameter("operation");
       session.setAttribute("myLineItems",
           myCart.getLineItems());

       if (myOperation.equals("Add")) {
           CartLineItem lineItem = new CartLineItem();
           lineItem.setId(request.getParameter("cdId"));
           lineItem.setCdtitle(request.getParameter("cdTitle"));
           lineItem.setPrice(request.getParameter("cdPrice"));
           myCart.lineItems.addElement(lineItem);
       }
```

Next, specify what happens when the current operation for the session is Delete. This happens when the user clicks the Delete button on the `Shopping Cart` page.

6. **Use the following code to specify what happens when the user clicks Delete:**

```
       if (myOperation.equals("Delete")) {
           String s = request.getParameter("cdId");
           System.out.println(s);
           int idVal = Integer.parseInt(s);
           myCart.removeLineItem(idVal);
       }
```

Finally, specify what happens when the Delete action deletes the last row of the `Cart` CD table. Use the JSP `forward` tag to go to the `EmptyCart` JSP page, which you will create soon. This last code ends the script you started in Step 5. You have to make a break for the `forward` tag, then resume before you finally end the scriptlet.

7. **Use the following code:**

```
        //If the last item is removed from the cart...
        if (((Vector)session.getAttribute("myLineItems")).size()
            == 0) {
            //End scriptlet temporarily so that you can use the JSP
            //"forward" tag to forward to the EmptyCart page.
            %>
            <jsp:forward page="EmptyCart.jsp" />
            //Resume the scriptlet.
            <%
        }
%>
```

Using Iteration Tags to Populate the `Cart` Table

Next, use the `forEach` and `out` tags to iterate through the passed data and fetch the field data of each row, much as you did in "Using the Iteration Tags to Display the Data" on page 52.

1. **Start a table for the purchase candidate data by creating the table headings:**

```
<TABLE border=1>
        <TR>
            <TH>ID</TH>
            <TH>CD Title</TH>
            <TH>Price</TH>
        </TR>
```

2. **Use the `forEach` and `out` tags to populate the table:**

```
<c:forEach var="item" items="${myLineItems}">
        <TR>
            <TD><c:out value="${item.id}"/></TD>
            <TD><c:out value="${item.cdtitle}"/></TD>
            <TD><c:out value="${item.price}"/></TD>
```

The `out` tags in the preceding code retrieve the value from the named fields in the current row of the results.

3. **Create a Delete button for each row, as in "Adding the Buttons to the Page" on page 70:**

```
<TD>
<form method=get action="ShopCart.jsp">
       <input type=hidden name=cdId value="<c:out value=
"${item.id}"/>">
       <input type=hidden name=cdTitle value="<c:out value=
"${item.cdtitle}"/>">
       <input type=hidden name=cdPrice value="<c:out value=
"${item.price}"/>">
       <input type=submit name=operation value="Delete">
</form>
</TD>
       </TR>
</c:forEach>
</TABLE>
```

Adding the Buttons to the Page

Finally, add the Resume Shopping, Place Order, and Cancel Order buttons to the page bottom.

- **Add the following code to the** `ShopCart` **JSP page:**

```
<p>
<!--Create the three buttons.-->
<form method=get action="ProductList.jsp">
       <input type=submit value="Resume Shopping">
</form>
<form method=get action="PlaceOrder.jsp">
       <input type=submit value="Place Order">
</form>
<form method=get action="CancelOrder.jsp">
       <input type=submit value="Cancel Order">
</form>
</p>
<!End the page.>
</body>
</html>
```

Tip – To see the completed source code of this page, see "`ShopCart.jsp` Source" on page 89.

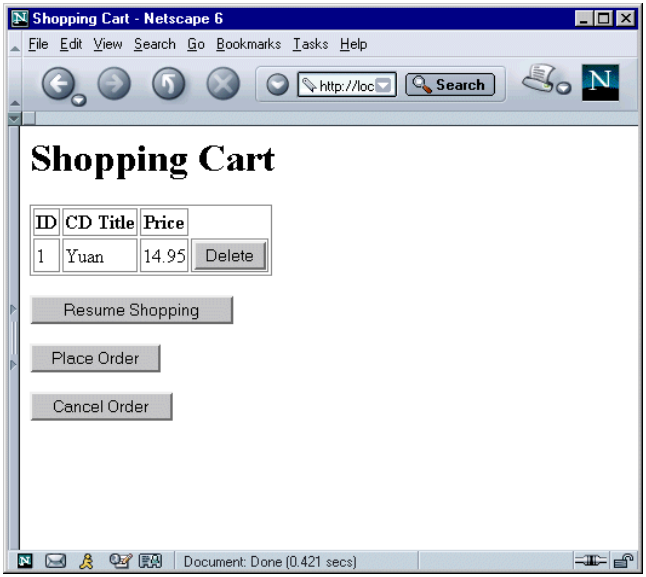
Testing the Shopping Cart Page

You do not test the `ShopCart` page directly. You test the `ProductList` page and navigate (by means of the Add button) to the Shopping Cart page.

1. **If not already started, start the PointBase Network Server by choosing Tools → PointBase Network Server → Start Server.**
2. **Select the `CDShopCart` web module and choose Build → Build All.**
3. **Right-click the `ProductList` JSP page and choose Execute (Force Reload).**

   After a few seconds, the CD Catalog List page is displayed.
4. **Click one of the Add buttons to navigate to the Shopping Cart page.**

   The Shopping Cart page looks similar to this example:



5. **Use the Resume Shopping button to return to the CD Catalog List page.**
6. **Terminate the execution by right-clicking the process in the Execution window and choosing Terminate Process.**
7. **Return to the editing workspace by clicking the Editing tab of the Explorer window.**

   You have almost finished the CDShopCart application. You only have to create the `EmptyCart` and `PlaceOrder` pages, and you're done.

Creating the Three Message Pages

Now create a JSP page that is displayed when a user empties the cart and two pages more that are displayed when the user clicks the Place Order and Cancel Order buttons.

In this section, perform the following tasks:

- "Creating the Empty Cart Page" which follows
- "Creating the Place Order Page" on page 73
- "Creating the Cancel Order Page" on page 75

Creating the Empty Cart Page

When an iterator tag finds an empty vector, it throws an exception rather than creating an empty table. You have dealt with this by testing for this case (Step 7 in "Adding Code to Add or Remove an Item From the Shopping Cart Table" on page 67) and then you handled the exception by displaying the Empty Cart page. This page contains a Resume Shopping button that enables the user to return to the product list page.

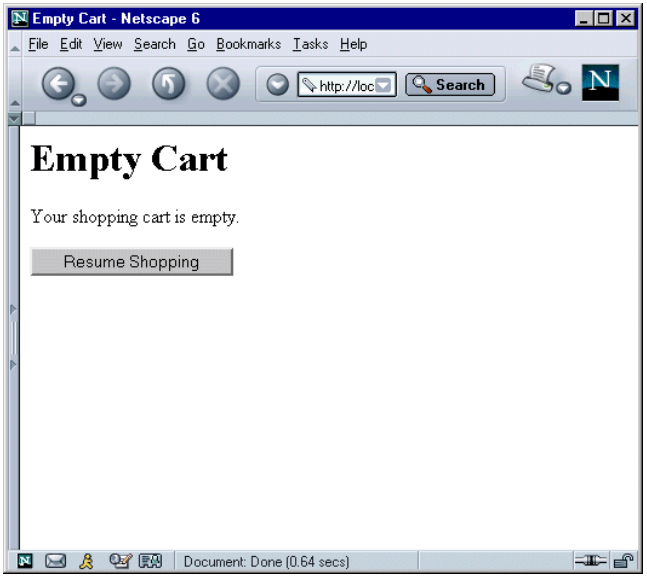


FIGURE 3-3 Empty Cart Page

To create the Empty Cart page:

1. **Right-click the** `CDShopCart` **web module and choose New → JSP & Servlet → JSP.**
2. **Type the name** `EmptyCart` **and click Finish.**

   The `EmptyCart` JSP page appears in the Explorer and its source in the Source Editor.
3. **Change the page title to** `Empty Cart` **and add code as follows:**

```
<%@page contentType="text/html"%>
<html>
<head><title>Empty Cart</title></head>
<body>
       <H1> Empty Cart </H1>
       <!--Display a message-->
       Your shopping cart is empty.
       <P>
       <!--Add a Resume Shopping button.-->
       <form method=get action="ProductList.jsp">
           <input type=submit value="Resume Shopping">
       </FORM>
       </P>
       </body>
</html>
```

4. **Save the** `EmptyCart` **page by choosing File → Save.**

Creating the Place Order Page

The Place Order and Cancel Order pages are very simple pages, and represent only one of many ways you can implement these actions. Because programming these pages demonstrates little more of the Sun ONE Studio 4 features than you have already seen, the tutorial uses the simplest possible implementation.

The Place Order page is displayed when the user clicks the Place Order button on the Shopping Cart page. Displaying this page ends the session.

FIGURE 3-4 Place Order Page

To create the Place Order page:

1. **Right-click the `CDShopCart` web module and choose New → JSP & Servlet → JSP.**
2. **Type the name `PlaceOrder` and click Finish.**

3. **Change the page title to Place Order and add code as follows:**

```
<%@page contentType="text/html"%>
<html>
<head><title>Place Order</title></head>
<body>
       <H1> Place Order </H1>
       <!--Invalidate the session-->
       <%
       session.invalidate();
       %>
       Your order has been placed. Thank you for shopping.
       <P>
       <FORM method=get action="ProductList.jsp">
           <INPUT type=submit value="Resume Shopping">
       </FORM>
       </P>
</body>
</html>
```

4. **Save the** `PlaceOrder` **page by choosing File → Save.**

Creating the Cancel Order Page

The Cancel Order page is displayed when the user clicks the Cancel Order button on the Shopping Cart page. Displaying this page ends the session. The page looks like FIGURE 3-5.

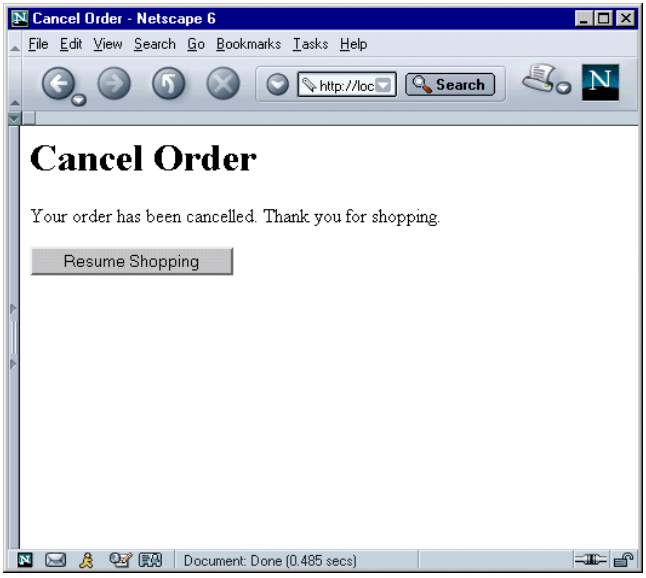


FIGURE 3-5 Cancel Order Page

To create the Cancel Order page:

1. **Right-click the** `CDShopCart` **web module and choose New → JSP & Servlet → JSP.**
2. **Type the name** `CancelOrder` **and click Finish.**
3. **Change the page title to** `Cancel Order` **and add code similar to the** `PlaceOrder` **page, as follows:**

```
<%@page contentType="text/html"%>
<html>
<head><title>Cancel Order</title></head>
<body>
       <H1> Cancel Order </H1>
       <%
       session.invalidate();
       %>
       Your order has been cancelled. Thank you for shopping.
       <P>
       <FORM method=get action="ProductList.jsp">
           <INPUT type=submit value="Resume Shopping">
       </FORM>
       </P>
</body>
</html>
```

4. **Save the `CancelOrder` page by choosing File → Save.**

Testing the Three Message Pages

As with the Shopping Cart page, test the message pages by running the `ProductList` JSP page. Then add CD items to the Shopping Cart and perform an appropriate action to display each message page.

1. **If not already started, start the PointBase Network Server by choosing Tools → PointBase Network Server → Start Server.**
2. **Select the `CDShopCart` web module and choose Build → Build All.**

   Everything should compile successfully.
3. **Right-click the `ProductList` JSP page and choose Execute (Force Reload).**

   Alternatively, you can right-click the `WEB-INF` node and choose Execute (Force Reload).

   After a few seconds, the CD Catalog List page is displayed.
4. **Click one of the Add buttons to navigate to the Shopping Cart page.**
5. **To test the Empty Cart page, click the Delete button on the item you just put into the cart.**

   The Empty Cart page appears.
6. **Click the Resume Shopping button to return to the CD Catalog List page.**
7. **Add one or more CDs to the cart.**
8. **Test the Cancel Order page by clicking the Cancel Order button.**
9. **When the page appears, click the Resume Shopping button to return to the CD Catalog List page.**
10. **Add another CD to the cart.**
11. **When the Shopping Cart page appears, make sure that the CD you added is the only one in the cart.**

    There should be only one CD in the cart because the Cancel Order action (Step 8) ended the previous session.
12. **Add more CDs to the cart, and then test the Place Order button.**
13. **When the Place Order page appears, click the Resume Shopping button to return to the Catalog page.**

14. **Add another CD to the cart.**

    The Place Order page ended the session. Therefore only one CD should be in the cart.

15. **To stop the application, right-click the message in the Execution window and choose Terminate Process.**

    This action terminates the Tomcat Web Server process.

APPENDIX **A**

CDShopCart Source Files

This appendix displays the code for the following CDShopCart components:

This code is also available as source files within the CDShopCart application zip file, which you can download from the Sun ONE Studio 4 Developer Resources portal at:

`http://forte.sun.com/ffj/documentation/tutorialsandexamples.html`

Tip – If you use these files to cut and paste code into the Sun ONE Studio 4 Source Editor, all formatting is lost. To automatically reformat the code, put the cursor in the Source Editor and press Ctrl-Shift-F.

ProductList.jsp Source

```
<%@page contentType="text/html"%>
<html>
<head><title>CD Catalog List</title></head>
<%@ taglib prefix="c" uri="http://java.sun.com/jstl/core" %>
<%@ taglib prefix="sql" uri="http://java.sun.com/jstl/sql" %>
<body>
<h1> CD Catalog List </h1>

<sql:setDataSource var="productDS"
       url="jdbc:pointbase:server://localhost/cdshopcart"
       driver="com.pointbase.jdbc.jdbcUniversalDriver"
       user="PBPUBLIC" />

<%--<sql:setDataSource var="productDS"
       url="jdbc:oracle:thin:@<hostname>:<port#>:<SID>"
       driver="oracle.jdbc.driver.OracleDriver"
       user="userid" /> --%>

<%--<sql:setDataSource var="productDS"
       url="jdbc:weblogic:mssqlserver4:<database>@<hostname>:<port#>"
       driver="weblogic.jdbc.mssqlserver4.Driver"
       user="userid" /> --%>

<sql:query var="productQuery"  dataSource="${productDS}"  >
       SELECT * FROM CD
</sql:query>

<TABLE border=1>
       <TR>
           <TH>ID</TH>
           <TH>CD Title</TH>
           <TH>Artist</TH>
           <TH>Country</TH>
           <TH>Price</TH>
       </TR>
```

```
<c:forEach var="row" items="${productQuery.rows}">
       <TR>
           <TD><c:out value="${row.ID}"/></TD>
           <TD><c:out value="${row.CDTITLE}"/></TD>
           <TD><c:out value="${row.ARTIST}"/></TD>
           <TD><c:out value="${row.COUNTRY}"/></TD>
           <TD><c:out value="${row.PRICE}"/></TD>
           <TD>
           <form method=get action="ShopCart.jsp">
               <input type=hidden name=cdId value="<c:out value="${row.ID}"/>">
               <input type=hidden name=cdTitle value="<c:out value=
"${row.CDTITLE}"/>">
               <input type=hidden name=cdPrice value="<c:out value=
"${row.PRICE}"/>">
               <input type=submit name=operation value=Add>
           </form>
           </TD>
       </TR>
</c:forEach>
</TABLE>
</body>
</html>
```

`CartLineItem` Bean Source

```
package ShopCart;

import java.beans.*;

public class CartLineItem extends Object implements java.io.Serializable {

       private static final String PROP_SAMPLE_PROPERTY = "SampleProperty";

       private String sampleProperty;

       private PropertyChangeSupport propertySupport;

       /** Holds value of property cdtitle. */
       private String cdtitle;
```

```
        /** Holds value of property id. */
        private int id;

        /** Holds value of property price. */
        private double price;

        /** Creates new CartLineItem */
        public CartLineItem() {
            propertySupport = new PropertyChangeSupport ( this );
        }

        public String getSampleProperty() {
            return sampleProperty;
        }

        public void setSampleProperty(String value) {
            String oldValue = sampleProperty;
            sampleProperty = valie;
            propertySupport.firePropertyChange(PROP_SAMPLE_PROPERTY, oldValue,
sampleProperty);
        }

        public void addPropertyChangeListener (PropertyChangeListener listener)
{
            propertySupport.addPropertyChangeListener (listener);
        }
        public void removePropertyChangeListener (PropertyChangeListener
listener){
            propertySupport.removePropertyChangeListener (listener);
        }

        /** Getter for property cdtitle.
        * @return Value of property cdtitle.
        */
        public String getCdtitle() {
            return this.cdtitle;
        }

        /** Setter for property cdtitle.
        * @param cdtitle New value of property cdtitle.
        */
        public void setCdtitle(String cdtitle) {
            this.cdtitle = cdtitle;
        }

        /** Getter for property id.
        * @return Value of property id.
        */
        public int getId() {
```

```
            return this.id;
        }

        /** Setter for property id.
        * @param id New value of property id.
        */
        public void setId(int id) {
            this.id = id;
        }

        /** Getter for property price.
        * @return Value of property price.
        */
        public double getPrice() {
            return this.price;
        }

        /** Setter for property price.
        * @param price New value of property price.
        */
        public void setPrice(double price) {
            this.price = price;
        }

        public void setId(java.lang.String pId) {
            int val = Integer.parseInt(pId);    // no error checking done here
            this.setId(val);
        }

        public void setPrice(java.lang.String pPrice) {
            double val = Double.parseDouble(pPrice);
            this.setPrice(val);
        }
}
```

CartLineItemBeanInfo Source

```
package ShopCart;

import java.beans.*;

public class CartLineItemBeanInfo extends SimpleBeanInfo {

       // Bean descriptor information will be obtained from introspection.
       private static BeanDescriptor beanDescriptor = null;
       private static BeanDescriptor getBdescriptor(){

           // Here you can add code for customizing the BeanDescriptor.

           return beanDescriptor;     }

       // Properties information will be obtained from introspection.
       private static PropertyDescriptor[] properties = null;
       private static PropertyDescriptor[] getPdescriptor(){
           if (properties == null) {
               try {
                   PropertyDescriptor[] props = {
                       new PropertyDescriptor("cdtitle", CartLineItem.class),
                       new PropertyDescriptor("id", CartLineItem.class),
                       new PropertyDescriptor("price", CartLineItem.class)};
                       properties = props;
               } catch (IntrospectionException ex) {
                   return null;
               }
           }
           return properties;     }

       // Event set information will be obtained from introspection.
       private static EventSetDescriptor[] eventSets = null;
       private static EventSetDescriptor[] getEdescriptor(){

           // Here you can add code for customizing the event sets array.

           return eventSets;     }

       // Method information will be obtained from introspection.
       private static MethodDescriptor[] methods = null;
```

```
        private static MethodDescriptor[] getMdescriptor(){

            // Here you can add code for customizing the methods array.

            return methods;     }

        private static int defaultPropertyIndex = -1;
        private static int defaultEventIndex = -1;


// Here you can add code for customizing the Superclass BeanInfo.

        /**
        * Gets the bean's <code>BeanDescriptor</code>s.
        *
        * @return BeanDescriptor describing the editable
        * properties of this bean.  May return null if the
        * information should be obtained by automatic analysis.
        */
        public BeanDescriptor getBeanDescriptor() {
            return getBdescriptor();
        }

        /**
        * Gets the bean's <code>PropertyDescriptor</code>s.
        *
        * @return An array of PropertyDescriptors describing the editable
        * properties supported by this bean.  May return null if the
        * information should be obtained by automatic analysis.
        * <p>
        * If a property is indexed, then its entry in the result array will
        * belong to the IndexedPropertyDescriptor subclass of PropertyDescriptor.
        * A client of getPropertyDescriptors can use "instanceof" to check
        * if a given PropertyDescriptor is an IndexedPropertyDescriptor.
        */
        public PropertyDescriptor[] getPropertyDescriptors() {
            return getPdescriptor();
        }

        /**
        * Gets the bean's <code>EventSetDescriptor</code>s.
        *
        * @return  An array of EventSetDescriptors describing the kinds of
        * events fired by this bean.  May return null if the information
        * should be obtained by automatic analysis.
        */
        public EventSetDescriptor[] getEventSetDescriptors() {
            return getEdescriptor();
        }
```

```
        /**
        * Gets the bean's <code>MethodDescriptor</code>s.
        *
        * @return  An array of MethodDescriptors describing the methods
        * implemented by this bean.  May return null if the information
        * should be obtained by automatic analysis.
        */
        public MethodDescriptor[] getMethodDescriptors() {
            return getMdescriptor();
        }

        /**
        * A bean may have a "default" property that is the property that will
        * mostly commonly be initially chosen for update by human's who are
        * customizing the bean.
        * @return  Index of default property in the PropertyDescriptor array
        *          returned by getPropertyDescriptors.
        * <P>  Returns -1 if there is no default property.
        */
        public int getDefaultPropertyIndex() {
            return defaultPropertyIndex;
        }

        /**
        * A bean may have a "default" event that is the event that will
        * mostly commonly be used by human's when using the bean.
        * @return Index of default event in the EventSetDescriptor array
        *          returned by getEventSetDescriptors.
        * <P>  Returns -1 if there is no default event.
        */
        public int getDefaultEventIndex() {
            return defaultEventIndex;
        }
}
```

`Cart` Bean Source

```
package ShopCart;

import java.beans.*;

public class Cart extends Object implements java.io.Serializable {

       private static final String PROP_SAMPLE_PROPERTY = "SampleProperty";

       private String sampleProperty;

       private PropertyChangeSupport propertySupport;

       /** Holds value of property lineItems. */
       public java.util.Vector lineItems;

       /** Creates new Cart */
       public Cart() {
           propertySupport = new PropertyChangeSupport ( this );
           lineItems = new java.util.Vector();
       }

       public String getSampleProperty () {
           return sampleProperty;
       }

       public void setSampleProperty (String value) {
           String oldValue = sampleProperty;
           sampleProperty = value;
           propertySupport.firePropertyChange (PROP_SAMPLE_PROPERTY, oldValue,
sampleProperty);
       }

       public void addPropertyChangeListener (PropertyChangeListener listener)
{
           propertySupport.addPropertyChangeListener (listener);
       }
```

```
        public void removePropertyChangeListener (PropertyChangeListener
listener){
            propertySupport.removePropertyChangeListener (listener);
        }

        /** Getter for property lineItems.
        * @return Value of property lineItems.
        */
        public java.util.Vector getLineItems() {
            return this.lineItems;
        }

        /** Setter for property lineItems.
        * @param lineItems New value of property lineItems.
        */
        public void setLineItems(java.util.Vector lineItems) {
            this.lineItems = lineItems;
        }

        /**
        * Returns the element number of the item in the cartItems as specified by
        * the passed ID
        */
        public int findLineItem(int pID) {
            System.out.println("Entering Cart.findLineItem()");
            int cartSize = (lineItems == null) ? 0 : lineItems.size();
            int i ;
            for ( i = 0 ; i < cartSize ; i++ )
            {
                if  ( pID == ((CartLineItem)lineItems.elementAt(i)).getId() )
                    break ;
            }

            if (i >= cartSize) {
                System.out.println("Couldn't find line item for ID: " + pID);
                return -1 ;
            }
            else
                return i ;
        }

        /** Removes a cartItem from the cartItems list.
        * @param pID ID of CartLineItem to remove.
        */
```

```
        public void removeLineItem(int pID) {
            System.out.println("Entering cart.removeLineItem()");
            int i = findLineItem(pID);
            if (i != -1) lineItems.remove(i);
            System.out.println("Leaving cart.removeLineItem()");
        }
}
```

ShopCart.jsp Source

```
<%@page contentType="text/html"%>
<%@ page import="java.util.*, ShopCart.*" %>

<html>
<head><title>Shopping Cart</title></head>
<%@ taglib prefix="c" uri="http://java.sun.com/jstl/core" %>
<body>
<h1> Shopping Cart </h1>

<jsp:useBean id="myCart" scope="session" class="ShopCart.Cart" />
<%
       String myOperation = request.getParameter("operation");
       session.setAttribute("myLineItems", myCart.getLineItems());

       if (myOperation.equals("Add")) {
           CartLineItem lineItem = new CartLineItem();
           lineItem.setId(request.getParameter("cdId"));
           lineItem.setCdtitle(request.getParameter("cdTitle"));
           lineItem.setPrice(request.getParameter("cdPrice"));
           myCart.lineItems.addElement(lineItem);
       }
       if (myOperation.equals("Delete")) {
           String s = request.getParameter("cdId");
           System.out.println(s);
           int idVal = Integer.parseInt(s);
           myCart.removeLineItem(idVal);
       }
       if (((Vector)session.getAttribute("myLineItems")).size() == 0) {
           %>
           <jsp:forward page="EmptyCart.jsp" />
```

```
            <%
        }
%>

<TABLE border=1>
<TR>
        <TH>ID</TH>
        <TH>CD Title</TH>
        <TH>Price</TH>
</TR>
<c:forEach var="item" items="${myLineItems}">
<TR>
        <TD><c:out value="${item.id}"/></TD>
        <TD><c:out value="${item.cdtitle}"/></TD>
        <TD><c:out value="${item.price}"/></TD>

        <TD>
            <form method=get action="ShopCart.jsp">
            <input type=hidden name=cdId value="<c:out value="${item.id}"/>">
            <input type=hidden name=cdTitle value="<c:out value=
"${item.cdtitle}"/>">
            <input type=hidden name=cdPrice value="<c:out value=
"${item.price}"/>">
            <input type=submit name=operation value=Delete>
            </form>
        </TD>
</TR>
</c:forEach>

</TABLE>
<p>
<form method=get action="ProductList.jsp">
        <input type=submit value="Resume Shopping">
</form>
<form method=get action="PlaceOrder.jsp">
        <input type=submit value="Place Order">
</form>
<form method=get action="CancelOrder.jsp">
        <input type=submit value="Cancel Order">
</form>
</p>

</body>
</html>
```

EmptyCart.jsp Source

```
<%@page contentType="text/html"%>
<HTML>
<head><title>Empty Cart</title></head>
<body>
       <h1> Empty Cart </h1>
       Your shopping cart is empty.
       <p>
       <form method=get action="ProductList.jsp">
           <input type=submit value="Resume Shopping">
       </form>
       </p>
</body>
</HTML>
```

PlaceOrder.jsp Source

```
<%@page contentType="text/html"%>
<html>
<head><title>Place Order</title></head>
<body>
       <h1> Place Order </h1>
       <%
       session.invalidate();
       %>
       Your order has been placed.  Thank you for shopping.
       <p>
       <form method=get action="ProductList.jsp">
           <input type=submit value="Resume Shopping">
       </form>
       </p>
</body>
</html>
```

CancelOrder.jsp Source

```
<%@page contentType="text/html"%>
<html>
<head><title>Cancel Order</title></head>
<body>
       <h1> Cancel Order </h1>
       <%
       session.invalidate();
       %>
       Your order has been cancelled. Thank you for shopping.
       <p>
       <form method=get action="ProductList.jsp">
           <input type=submit value="Resume Shopping">
       </form>
       </p>
</body>
</html>
```

APPENDIX **B**

CDShopCart Database Scripts

This appendix displays the following database scripts for the CDShopCart tutorial:

These scripts are also available as files within the CDShopCart application zip file, which you can download from the Sun ONE Studio 4 Developer Resources portal at:

`http://forte.sun.com/ffj/documentation/tutorialsandexamples.html`

Script for a PointBase Database

Use the following SQL script with a PointBase database:.

```
drop table CD;

create table CD(
       id     int,
       cdtitle char(20),
       artist  char(20),
       country char(20),
       price   number(8,2),
       primary key(id));

insert into CD (id, cdtitle, artist, country, price)
  values (1, 'Yuan','The Guo Brothers', 'China',14.95);
insert into CD (id, cdtitle, artist, country, price)
  values (2, 'Drums of Passion','Babatunde Olatunji',
'Nigeria',16.95);
insert into CD (id, cdtitle, artist, country, price)
  values (3, 'Kaira','Tounami Diabate', 'Mali',13.95);
insert into CD (id, cdtitle, artist, country, price)
  values (4, 'The Lion is Loose','Eliades Ochoa', 'Cuba',12.95);
insert into CD (id, cdtitle, artist, country, price)
  values (5, 'Dance the Devil Away','Outback', 'Australia',14.95);

commit;
```

Script for an Oracle Database

Use the following SQL script with an Oracle database:

```
//* Run with: sqlplus tutorial/tutorial@dbname @scriptname */
drop table CD;

create table CD(
       id     int,
       cdtitle char(20),
       artist  char(20),
       country char(20),
       price   number(8,2),
       primary key(id));
grant all on CD to public;

insert into CD (id, cdtitle, artist, country, price)
  values (1, 'Yuan','The Guo Brothers', 'China',14.95);
insert into CD (id, cdtitle, artist, country, price)
  values (2, 'Drums of Passion','Babatunde Olatunji',
'Nigeria',16.95);
insert into CD (id, cdtitle, artist, country, price)
  values (3, 'Kaira','Tounami Diabate', 'Mali',13.95);
insert into CD (id, cdtitle, artist, country, price)
  values (4, 'The Lion is Loose','Eliades Ochoa', 'Cuba',12.95);
insert into CD (id, cdtitle, artist, country, price)
  values (5, 'Dance the Devil Away','Outback', 'Australia',14.95);
commit;
```

Script for a Microsoft SQLServer Database

Use the following SQL script with a Microsoft SQLServer database:

```
use cdcat
go
drop table CD
go

create table CD(
       id     int,
       cdtitle varchar(20) null,
       artist  varchar(20) null,
       country varchar(20) null,
       price   money null,
PRIMARY KEY(id))
go
grant all on CD to public
go

insert into CD (id, cdtitle, artist, country, price)
  values (1, 'Yuan','The Guo Brothers', 'China',14.95)
insert into CD (id, cdtitle, artist, country, price)
  values (2, 'Drums of Passion','Babatunde Olatunji',
'Nigeria',16.95)
insert into CD (id, cdtitle, artist, country, price)
  values (3, 'Kaira','Tounami Diabate', 'Mali',13.95)
insert into CD (id, cdtitle, artist, country, price)
  values (4, 'The Lion is Loose','Eliades Ochoa', 'Cuba',12.95)
insert into CD (id, cdtitle, artist, country, price)
  values (5, 'Dance the Devil Away','Outback', 'Australia',14.95)
go
```

Script for an IBM DB2 Database

Use the following SQL script with an IBM DB2 database:

```
drop table CD

create table CD(
       id     int not null primary key ,
       cdtitle char(20),
       artist  char(20),
       country char(20),
       price   num(8,2))

insert into CD (id, cdtitle, artist, country, price)
   values (1, 'Yuan','The Guo Brothers', 'China',14.95)
insert into CD (id, cdtitle, artist, country, price)
   values (2, 'Drums of Passion','Babatunde Olatunji',
'Nigeria',16.95)
insert into CD (id, cdtitle, artist, country, price)
   values (3, 'Kaira','Tounami Diabate', 'Mali',13.95)
insert into CD (id, cdtitle, artist, country, price)
   values (4, 'The Lion is Loose','Eliades Ochoa', 'Cuba',12.95)
insert into CD (id, cdtitle, artist, country, price)
  values (5, 'Dance the Devil Away','Outback', 'Australia',14.95)
```

Index

D

E

F

H

I

J

L

U

V

W