Lab Appendix

Guidelines for Configuring the Course IDE - Forte™ for Java™, Community Edition

Use the information provided below to configure the Forte for Java, Community Edition (CE) IDE's to allow components executing in the IDE to access applications deployed to the J2EE runtime environment.

Establish the Working Environment:

- Configure the following environment variables:
  - `J2EE_HOME` - The path to the J2EE installation directory; for example, `d:\j2sdkee1.2.1`.
  - `JAVA_HOME` - The path to the JDK installation directory; for example, `d:\jdk1.3`.
  - `CLASSPATH` - Add the paths `%J2EE_HOME%\lib\j2ee.jar;.;%JAVA_HOME%\lib\tools.jar`
- Create a home directory for placing the files generated by the J2EE deployment tool; for example, `d:\J2EEAppHome`.

Creating a J2EE Execution Setting:

The execution setting determines the execution environment invoked when executing a Java program from within the Forte for Java IDE. From the Forte for Java User Guide:

"An application run using internal execution runs inside the Forte for Java IDE. This brings the advantages that the application can modify the IDE itself and be loaded faster. But it imposes at least two restrictions on the executed application. The application cannot install its own `URLStreamHandlerFactory` or `SecurityManager` (so you cannot run RMI applications, for example). In addition, if the executed application crashes, the IDE crashes with it."

"Most applications use external execution, and it is set in most of the templates that come with the IDE. A new virtual machine (VM) is invoked for executing the application. This enables you to run applications that require a special VM or need to do operations that aren't possible with internal execution (see below). You can specify the VM executable (such as `java.exe`) and complete command line parameters together with class path settings for the application. External execution also protects the IDE from application crashes and delays."

To access components deployed to the J2EE runtime from a CE client you must configure an External Execution Setting that mimics starting the client from the command line as described below.

- Create a new External Execution named J2EE Execution (right click on Project > Settings > ExecutionTypes > ExternalExection and choose New External Execution Service – refer to the Forte for Java user's Guide for more information).
- On the Expert tab for the new execution setting configure the following items (Press the Enter key after configuring a property in Forte for Java in order to save the settings).
  - Boot Class Path: The directory where `jdk1.3` is installed (for `example d:\jdk1.3`).
  - Class Path: add the following paths:
    - The path to the `j2ee.jar` file
    - The path to the client `.jar` created during deployment
    - The current directory
    - The path to the Forte for Java `Development` directory
    - For example:
      `d:\j2sdkee1.2.1\lib\j2ee.jar;d:\J2EEAppHome\MyAppClient.jar;.;d:\forte4jCE\Development`
- On the Properties tab, open the External Process window, and configure the following item:
  - Arguments: `-cp {classpath} {classname} {arguments}`
- Right-click the client class in the Forte for Java IDE Filesystem Explorer, choose properties, select the Execution tab, and configure the Executor property to use the newly created J2EE Execution.

Guidelines for Configuring the Course IDE - Forte for Java Community Edition (continued)

Installing Templates:

Forte for Java uses a New From Template wizard to assist in the creation of simple Java classes. A template is a `.java` file that contains some Forte for Java keywords used to insert some required information generated by the wizard, such as class name, creator, and creation date. You can add your own templates to the IDE to supplement the selection provided with the tool. The lab resources include a set of templates used to create EJB classes and interfaces as well as a data model bean. To install the templates, you must create a new template folder for the set of EJB templates and then install the suplied EJB templates into the new EJB templates package. You also need to install the `DataModelBean` template into the existing `Beans` package. Use the instructions described below to complete this process. For more information, search on "Creating your own templates" and "Modifying existing templates" in the IDE's online help system.

- To add a new template folder:
  - Choose Tools > Global Options from the command console menu.
  - Right click on the Templates label and choose New Package.
  - Enter the package name, for example "EJB", and click OK.
  - Expand the Templates hierarchy and verify that the new Package has been added.
- To add a template to an existing template folder:
  - To add a `.java` file to the template hierarchy, it must first be visible from within the IDE. To mount the file system containing the new templates:
    - Click the Editing tab on the command console to open up the Editing workspace.
    - Mount the directory that contains the `.java` template files by right-clicking on the Filesystems icon in the IDE's explorer window and choosing Mount Directory.
    - Browse to the directory containing the template files you want to add, and then choose Mount.
  - Expand the file system in the IDE explorer window that contains the new template files.
  - Right-click on a template file, and choose Save As Template.
  - Select the desired template package from the dialog box, and choose OK.
  - The template appears as an option when choosing New From Template.

Guidelines for Using the Course IDE - Forte for Java, Community Edition

IDE Workspaces:

The Forte for Java IDE has a suite of workspaces to organize the visible windows based on the required functionality. The available workspaces are Editing, GUI Editing, Browsing, Running, and Debugging. See the IDE help system for a complete explanation of the functionality provided by each of these workspace views and for instructions on configuring the workspace view. You can switch between workspaces using the tabs on the main command console.

Use the Editing workspace to create your application components for the course lab exercises. The IDE automatically switches to the Running workspace when you execute your test client class. Reselect the Editing tab from the command console after running your application to return to the Editing workspace.

The Explorer Window:

The Explorer in Forte for Java, Community Edition gives a unified view of all objects and files and provides a starting point for many programming functions. Here, you can work with objects, organize your work, modify object properties, and connect various data sources. When you first start Forte for Java and open the Explorer, you will see a multi-tab window with tabs labeled Filesystems, Project, Runtime, and Javadoc. Click on a tab to go to that part of the Explorer.

Select the Filesystems tab to display the file system view in which you create your application components.

Creating Application Components:

In Forte for Java, Community Edition, you create new classes (as well as other objects) with templates. The template serves as a skeleton for the class and includes basic source code for that class. If you prefer to write all of the code yourself, you can choose the Empty template, and a file will be generated with the only code being the name of the package where you have created the template. A wide range of templates come with the IDE, and you can also create your own.

Classes are created using the New From Template wizard. You can access this wizard from the main window, the Explorer, and the Object Browser. When you access the wizard from the main window, you select the type of object you want to create from the first page of the wizard, which also provides short descriptions of each template. When you access the wizard from a contextual menu in the Explorer or the Object Browser, you skip straight to the second page of the wizard. In addition, the package information is automatically entered for you, based on where you right-clicked. The process for creating a class from within the IDE Explorer window is described below.

The following templates have been added to the IDE for this course:

- Beans
  - DataModelBean
- EJBs
  - EntityEJB (entity EJB bean class)
  - EntityHomeInterface
  - RemoteInterface (EJBObject, identical for session and entity EJBs)
  - SessionEJB (session EJB bean class)
  - SessionHomeInterface

Guidelines for Using the Course IDE - Forte for Java, Community Edition (continued)

To create a new Java package from the Explorer [Filesystems] window:

- If the Explorer is not open, choose File > Open Explorer from the main menu and click on the Filesystems tab to open the file system view.
- Select the branch of the filesystem in which to create the new package. Create your packages in the `<FORTE_HOME>/Development` directory for these lab exercises.
- Right click on selected files ystem, for example `<FORTE_HOME>/Development`, and choose New Package.
- Enter the package name in the Create New Package dialog box and select OK.
- The new package will appear as a branch under the selected file system.

To create an object (class or interface) from the Explorer [Filesystems] window:

- If the Explorer is not open, choose File > Open Explorer from the main menu and click on the Filesystems tab to open the file system view
- Find the package (marked with a folder icon) under the Filesystems tab in the Explorer where you want to place the class and right-click on the package to bring up its contextual menu
- Choose New From Template, the template type from the first submenu, and then the template itself from the second submenu
- The Target Location page of the New From Template wizard will appear
- Type the name of the object in the Name field, verify that the correct package is selected in the Package panel, and click Finish

To create bean fields with associated accessor and mutator methods:

- In the Explorer [Filesystems] window, expand the branch containing the class in which you want to add the new field
- Right click on Bean Patterns and choose New > Property
- Enter the field Name and Type
- Select the desired accessibility level (Read/Write, Read Only, Write Only)
- Select Generate field to create the field element in the class
- Select Generate return statement (for Read/Write or Read Only fields) to generate an accessor method for the field
- Select Generate set statement (for Read/Write or Write Only fields) to generate a mutator method for the field
- Click OK to add the property, the field, and the associated methods to the class

To mount a .jar file into the IDE's filesystem:

- If the Explorer is not open, choose Open Explorer from the File menu or toolbar and click on the Filesystems tab to open the filesystem view
- In the Explorer [Filesystems] window, right click on Filesystems and select Add JAR to display the Add JAR Archive dialog box
- Navigate to the location of the `.jar` file, select the file and choose Add
- A mounted `.jar` file appears as a branch in the filesystem

Guidelines for Using the Course IDE - Forte for Java, Community Edition (continued)

To add a directory to the IDE's filesystem:

- If the Explorer is not open, choose Open Explorer from the File menu or toolbar and click on the Filesystems tab to open the file system view
- In the Explorer [Filesystems] window, right click on Filesystems and select Add Directory to display the Add Directory dialog box
- Navigate to the desired directory, select the directory and choose Add

Editing Java Sources

The Editor is a full-featured text editor that is integrated with the Form Editor, Explorer, Compiler, and Debugger. It is the default viewer for all Java, HTML, and plain text files as well as any other types of files specified by installed extension modules. Each of these open files is displayed as a page in a multi-tab window – a single window with multiple tabs enabling you to choose which page to view in the window by clicking one of the tabs. Advanced features include customizable abbreviations and dynamic Java code completion.

The Editor contains source code that is syntactically colored. Different colors signify different text properties. For example, by default all keywords are shown in blue and all comments in light gray. Guarded text generated by the Form Editor has a blue background by default and cannot be edited. The bottom of the Editor window has one or more tabs used to view different documents. From each of these tabs, the document can be saved, closed, docked or undocked, and cloned.

Double-click a Java or text object in the Explorer or the Object Browser to open the file in the Editor (or select the node of the object and press the `ENTER` key). Any files that you subsequently open will also appear in the Editor with their own separate tabs in the bottom of the window. The tab of the currently visible file is highlighted. To flip between displayed files, simply click the tab of the file you want displayed or use the `Alt+LEFT` and `Alt+RIGHT` keyboard shortcuts.

The Editor uses standard mouse functions – click and drag the left mouse button to select a block of text, double-click the left button within a word to select that word, and triple-click to select an entire line. You can also select a block of text by clicking to place the insertion point where the block should begin, holding down `SHIFT`, and then clicking to determine the end of the selection.

You can select text and move it to and from the clipboard using the Cut, Copy, Paste, and Delete commands, which are available in the Edit menu, on the Edit toolbar, and in the contextual menu (accessed by right-clicking on the selected text). You can use Undo to reverse the previous command and Redo to reverse the reversal. These commands are available in the Edit menu, on the Edit toolbar, in various contextual menus, and by using keyboard shortcuts.

Forte for Java, Community Edition also has a dynamic code completion feature, where you can type a few characters and then bring up a list of possible classes, methods, variables, and so on that can be used to complete the expression.

To use Java code completion:

- Type the first few characters of the expression (for example, `import javax.` or `someFile.getP`)
- Press `CTRL+SPACE` (or pause after entering a period)
- When the code completion box appears use the most convenient combination of the following steps:
  - Keep typing to narrow down the selection of items in the list
  - Use the navigation keys (`UP` arrow, `DOWN` arrow, `PAGE-UP`, `PAGE-DOWN`, `HOME`, and `END`) or the mouse to scroll through the list and select an expression
  - Press the `ENTER` key to enter the selected method into your file and close the code completion box
  - Press `TAB` to select the longest common sub string matching the text you have typed (Bash-style context-sensitive completion) and keep the list box open

Guidelines for Using the Course IDE - Forte for Java, Community Edition (continued)

Compiling Java Sources:

Forte for Java, Community Edition offers a wide array of compilation options, from different ways to bring up the Compile command to the ability to use different compilers and set a specific compiler for each class.

Note: When you choose the Compile (or Compile All, Compile Project, Build, Build All, or Build Project) command for an object, the IDE (consistent with Java conventions) automatically compiles the first file it finds with the same name and package. Therefore, if you have two files with the same file name and package hierarchy mounted in the Explorer, the file in the first package listed will be compiled automatically, even if you choose the Compile command with the second package selected.

After you initiate compilation of a class, the Output Window appears if there are any compilation errors. If you double-click on the error in the Output Window, the insertion point will jump to the line in the Editor with the error.

Compilation progress is reported in the status line (next to the workspace tabs in the main window).

To compile an object:

- Right click on the object (class file or package) in the Explorer window
- Choose the desired compilation option from the submenu (Compile, Compile All, Build, or Build All)
    - Choosing Compile when a folder is selected compiles all sources in that folder which have been modified since they were last compiled or that have not been previously compiled
    - Choosing Compile All does this recursively on a folder and all its sub-folders
    - Choosing the Build command (also available with keyboard shortcut ALT+F9) is slightly different from compiling in that it forces re-compilation of all sources in a folder, whether they are current or not. Use this option when you wish to be sure that all of your code can compile together
    - Choosing Build All recursively builds a folder and all of its sub-folders

To run a Java application:

- Make sure that the Java object is executable (that it either has a `main` method or is a subclass of `Applet` or `JApplet`)
- Right-click on the object in the Explorer and choose Execute from the contextual menu

When executing, the Java class is (by default) first compiled. After compilation completes successfully, the IDE switches to the Running Workspace (though you may configure it to do otherwise by opening the Project Settings window, selecting the Execution Settings node, and changing the Workspace property). You can return the IDE to the Editing or the GUI Editing workspace by clicking the tab for the workspace in the lower left part of the Main Window.

Guidelines for Creating a Session EJB

Creating a session EJB involves creating both the home and remote interfaces and a bean class using the general guidelines listed below.

The Home Interface:

- Extends `EJBHome` and serves as a factory for session bean instances
- Defines session beans `create` methods corresponding to `ejbCreate` methods in the bean class
  - The home interface creator methods return the remote interface type of the EJB
  - A stateless session EJB cannot have parameterized creator methods
- Method definitions throw:
  - `java.rmi.RemoteException` (required)
  - `javax.ejb.CreateException` (required)

The Remote Interface:

- Extends `EJBObject` and is a reference to a bean instance
- Contains method stubs that match business methods implemented in the bean class
- Method definitions throw:
  - `java.rmi.RemoteException` (required)
  - application exceptions (as needed)

The Bean Class:

- Implements `SessionBean` as the bean instance
- Requires a default constructor
- Declares `ejbRemove`, `ejbActivate`, `ejbPassivate`, and `setSessionContext` methods and implements them if required
  - No method stubs are required for these methods on the home or remote interfaces
  - These methods are called by the container when needed
- Implements one or more `ejbCreate` methods corresponding to the `create` methods defined on the home interface
  - Name: `ejbCreate`
  - Visibility: `public`
  - Return: `void`
  - Parameters: Must match parameters on the corresponding method stub defined on the home interface (Note that a stateless session EJB cannot have parameterized creator methods)
  - Methods throw:
    - `javax.ejb.CreateException`

Guidelines for Creating a Session EJB (continued)

The Bean Class (continued):

- Implements business methods defined on the remote interface:
  - Visibility: `public`
  - Return: Method-specific
  - Methods throw:
    - `javax.ejb.EJBException` to indicate system level problems; the container wraps in a `RemoteException` so it does not need to be declared in the throws clause
    - Application exceptions (as needed)

Guidelines for Creating a BMP Entity EJB

Creating a BMP entity EJB involves creating both the home and remote interfaces and a bean class using the general guidelines listed below.

The Home Interface:

- Extends `EJBHome` and serves as a factory for entity bean instances
- Defines entity beans creator and finder methods corresponding to the `ejbCreate` and `ejbFind` methods in the bean class
  - The home interface method declarations return the remote interface type of the EJB
  - The creator methods on an entity bean add a row to the data store
    - Creator methods are not required on beans that do not add rows to the data store
  - Finder methods are used to locate a row or set of rows in the data source
    - An entity bean's home interface must define a `findByPrimaryKey` method implemented by the bean class's `ejbFindByPrimaryKey`
    - Other finder methods can be defined and implemented as required
- Method definitions throw:
  - `java.rmi.RemoteException` (required)
  - `javax.ejb.CreateException` (required for creator methods)
  - `javax.ejb.DuplicateKeyException` (required for creator methods)
  - `javax.ejb.FinderException` (required for finder methods)

The Remote Interface:

- Extends `EJBObject` and is a reference to a bean instance
- Contains method stubs that match business methods implemented in the bean class
- Method definitions throw:
  - `java.rmi.RemoteException` (required)
  - Application exceptions (as needed)

The Bean Class:

- Implements `EntityBean` as the bean instance
- Requires a default constructor
- Defines persistent fields that map to corresponding columns in the data store
- Defines `ejbActivate`, `ejbPassivate`, `setEntityContext`, and `unsetEntityContext` and implements them, if required
  - No method stubs are required for these methods on the home or remote interfaces
  - These methods are called by the container when needed

Creating a BMP entity EJB involves creating both the home and remote interfaces and a bean class using the general guidelines listed below.

Guidelines for Creating a BMP Entity EJB (continued)

The Bean Class (continued):

- Implements `ejbCreate` methods corresponding to the `create` methods defined on the home interface – an entity EJB does not require a creator method
  - Name: `ejbCreate`
  - Visibility: `public`
  - Return: `void`
  - Parameters: Must match parameters on the corresponding method stub defined on the home interface
  - Methods throw:
    - `javax.ejb.CreateException`
    - `javax.ejb.DuplicateKeyException`
- Implements the life-cycle methods `ejbLoad` and `ejbStore` used to synchronize the bean's state with the data store
  - Visibility: `public`
  - Return: `void`
  - Methods throw:
    - `javax.ejb.EJBException` (does not need to be declared in the throws clause)
- Declares and optionally implements `ejbRemove` which is called by the container when needed – no method stub is required for this method on the home or remote interface
  - Visibility: `public`
  - Return: `void`
  - Method throws:
    - `javax.ejb.RemoveException`
- Implements the finder methods defined by the home interface – an entity bean must implement the `ejbFindByPrimaryKey` method
  - Visibility: `public`
  - Return: The primary key type
  - Parameters: Must match parameters on the corresponding method stub defined on the home interface
  - Methods throw:
    - `javax.ejb.FinderException`

Guidelines for Creating a BMP Entity EJB (continued)

The Bean Class (continued):

- Implements business methods defined on the remote interface with the following characteristics:
  - Visibility: `public`
  - Return: Method-specific
  - Parameters: Must match parameters on the corresponding method stub defined on the remote interface
  - Methods throw:
    - `javax.ejb.EJBException` to indicate system level problems – the container wraps in a `RemoteException` so it does not need to be declared in the throws clause
    - Application exceptions (as needed)

The Primary Key Class:

- Defines the EJB's primary key that is used by the container to uniquely identify a bean instance
- Must implement `equals` and `hashCode` methods
- Create a EJB-specific primary key class for complex primary keys or key classes that do not implement the `equals` and `hashCode` methods

Guidelines for Using BMP Entity EJBs With Data Access Objects

The code required in each EJB method that uses a method on a Data Access Object takes the following form:

- Instantiate a local instance of the DAO; for example,

  ```
  SampleDAO SDAO = new SampleDAO();
  ```

- Use the DAO's mutator methods to initialize the required persistent fields. In most cases, you want the DAO's primary key field to match the primary key of the EJB instance. You can retrieve the value of the EJB's primary key using the EJB's *<EntityContext>*`.getPrimaryKey` method; for example,

  `SDAO.set`*<key_field>*`((`*<Key_Type>*`)context.getPrimaryKey());`

  In the case where you are invoking the DAO's `findByPrimaryKey` method, you set the DAO's primary key field to the value of the "key" argument passed to `ejbFindByPrimaryKey`; for example:

  `SDAO.set`*<key_field>*`(key);`

  In the case where you are invoking the DAO's `store` method, you must set all fields on the DAO that map to the persistent fields on the EJB.

- Start a try block, and:
  - Use the EJB's `getDBConnection` method to establish database connection; for example:

    ```
    dbConnection = getDBConnection();
    ```

  - Invoke the appropriate method on the DAO passing the database connection as an argument; for example:

    `SDAO.load(dbConnection)` or `SDAO.remove(dbConnection)`, and so on.

- Catch `java.sql.SQLException` and throw `RemoveException` for `ejbRemove`, `FinderException` for `ejbFindByPrimaryKey`, and `EJBException` for all other methods.
- Start a finally block and...
  - Start a try block and...
    - Close the database connection; for example:

      ```
      dbConnection.close();
      ```

  - Catch `java.sql.SQLException` and throw `RemoveException` for `ejbRemove`, `FinderException` for finder methods, and `EJBException` for all other methods.
- If necessary, set the EJB's local persistent fields to match the state of the DAO. This is only required when invoking the `load` method in this application.

For example, the code for an `ejbRemove` method might look like this:

```
public void ejbRemove() throws RemoveException {
   InventoryDAO IDAO = new InventoryDAO();
   IDAO.setItemId((String)context.getPrimaryKey());
   try{
      dbConnection = getDBConnection();
      IDAO.remove(dbConnection);
   }catch (java.sql.SQLException se) {
      throw new RemoveException ("SQL Exception in remove");
   } finally {
      try {
         dbConnection.close();
      } catch (SQLException se) {
         throw new RemoveException ("SQL Exception in remove");
      }
   }
}
```

Guidelines for Creating a CMP Entity EJB

A CMP Entity EJB has a structure identical to a BMP entity EJB and, therefore, has similar requirements and specifications. The difference between CMP and BMP entity EJBs involves the container's role in providing functionality for the CMP bean, primarily in the form of database access and data manipulation routines that otherwise must be implemented by the bean developer when creating a BMP entity bean. Eliminating the need to code the data access routines results in a considerable work savings when coding the CMP entity EJB's bean class. Consequently, coding a CMP entity bean involves less effort then implementing the same functionality using a BMP entity EJB. However, when compared to a BMP entity EJB, a CMP entity EJB normally requires additional configuration during assembly and deployment to resolve the database connectivity settings and flesh out the SQL syntax used by the container to interact with the data store.

Creating a CMP entity EJB involves creating both the home and remote interfaces and a bean class using the general guidelines listed below.

The Home Interface:

- Extends `EJBHome` and serves as a factory for entity bean instances
- Defines entity beans creator and finder methods corresponding to the `ejbCreate` and `ejbFind` methods in the bean class
  - The home interface method declarations return the remote interface type of the EJB
  - The creator methods on an entity bean add a row to the data store
    - Creator methods are not required on beans that do not add rows to the data store
    - Creator method arguments are normally used to initialize the state of the created entity bean
  - Finder methods are used to locate a row or set of rows in the data source
    - An entity bean's home interface must define a `findByPrimaryKey` method
    - Other finder methods are defined and implemented as required
- Method definitions throw:
  - `java.rmi.RemoteException` (required)
  - `javax.ejb.CreateException` (required for creator methods)
  - `javax.ejb.DuplicateKeyException` (required for creator methods)
  - `javax.ejb.FinderException` (required for finder methods)

The Remote Interface:

- Extends `EJBObject` and is a reference to a bean instance
- Contains method stubs that match business methods implemented in the bean class
- Method definitions throw:
  - `java.rmi.RemoteException` (required)
  - Application exceptions (as needed)

The Bean Class:

- Implements `EntityBean` as the bean instance
- Requires a default constructor
- Defines persistent fields that map to corresponding columns in the data store – persistent fields must have public visibility so the container can access them as needed

Guidelines for Creating a CMP Entity EJB (continued)

The Bean Class (continued):

- Defines `ejbActivate`, `ejbPassivate`, `setEntityContext`, and `unsetEntityContext` and implements them, if required
  - No method stubs are required for these methods on the home or remote interfaces
  - These methods are called by the container when needed
- Implements `ejbCreate` methods corresponding to the `create` methods defined on the home interface
  - Name: `ejbCreate`
  - Visibility: `public`
  - Return: `void`
  - Parameters: Must match parameters on the corresponding method stub defined on the home interface
  - Methods throw:
    - `javax.ejb.CreateException`
    - `javax.ejb.DuplicateKeyException`
  - In most cases the only functionality a bean developer adds to the `ejbCreate` method checks the validity of the primary key and initializes the EJB's persistent fields with the input parameters
  - The container generates the SQL commands required to add a new row to the data store
  - An entity EJB does not require a creator method
- Declares `ejbRemove`
  - Visibility: `public`
  - Return: `void`
  - Method throws:
    - `javax.ejb.RemoveException`
  - The container generates the SQL commands required to delete a row from the data store
  - No method stub is required for this method on the home or remote interface
  - `ejbRemove` is called by the container when needed
- Declares the life-cycle methods `ejbLoad` and `ejbStore` used to synchronize the bean's state with the data store – the container generates the SQL code required to implement these methods
- Finder methods defined by the home interface are not implemented in the EJB's bean class but are, instead, configured when the application is assembled and deployed

Guidelines for Creating a CMP Entity EJB (continued)

The Bean Class (continued):

- Implements business methods defined on the remote interface:
  - Visibility: `public`
  - Return: Method-specific
  - Parameters: Must match parameters on the corresponding method stub defined on the remote interface
  - Methods throw:
    - `javax.ejb.EJBException` to indicate system level problems – the container wraps it in a `RemoteException` so it does not need to be declared in the throws clause
    - Application exceptions (as needed)

The Primary Key Class:

- Defines the EJB's primary key, which is used by the container to uniquely identify a bean instance
- Can be any class that is a legal RMI-IIOP value yype
- Must implement `equals` and `hashCode` methods
- Create a EJB-specific primary key class for complex primary keys or key classes that do not implement the `equals` and `hashCode` methods

Assembly and Deployment Guidelines for J2SDKEE 1.2.1 Reference Implementation

The steps required to assemble and deploy J2EE components and applications in the JSDKEE1.2.1 Reference Implementation are described below. These instructions assume that you are using the assembly/deployment tools and Cloudscape database provided with the SDK.

Starting the tools:

- Start the J2EE runtime, J2EE assembly/deployment tool, and the Cloudscape database using the commands as described in the SDK documentation. You can start/stop each of these J2EE component using the following commands from the `<J2EEHome>/bin` directory:

To start the J2EE runtime use:	`j2ee -verbose`
To start the Cloudscape database use:	`cloudscape -start`
To start the assembly/deployment tool use:	`deploytool`
To stop the Cloudscape database use:	`cloudscape - stop`
To stop the J2EE runtime use:	`j2ee - stop`

- After the J2EE tools and runtime environment are up and running, you can proceed to assemble, deploy, and run your J2EE application.

Creating a J2EE application:

- Choose File > New Application from the Deployment Tool Main Menu.
- In the New Application dialog box, to enter the **Application File Name**, browse to the directory where you would like the application ear file stored and enter a name for the new J2EE application using an `.ear` extension; for example, `C:\Forte4JCE\J2EE\BankApp.ear`.
- Enter a name into the **Application Display Name** field to display in the tool's browser; for example, `BankApp`. By default, the deployment tool creates the Display Name for you based on the Application File Name.
- Choose OK to create the new application.

Adding an EJB to a J2EE application:

- Choose File > New Enterprise Bean from the Deployment Tool Main Menu. The Help button displayed on each dialog box for the New Enterprise Bean Wizard invokes the help system with helpful information about the data elements contained on each screen. A brief explanation of each of the EJB Wizard's dialog boxes is given below. Consult the help documentation for more detailed information about specific screens and data elements.
- **New Enterprise Bean Wizard - EJB JAR** dialog box:
  - Identify the location for the EJB, create a `.jar` file if necessary, and select the EJB's class files
    - Note: Class files should be chosen from packages contained within the selected root directory. For example, choose the root directory `D:\Forte4jCE\Development`, open the EJB package, select the class files, and choose Add.
- **New Enterprise Bean Wizard - General** dialog box:
  - Select the appropriate class files for the EJB's Enterprise Bean Class, home Interface and remote interface, enter a display name and description for the EJB, and define the EJB type: Entity or Session (stateful or stateless)
- **New Enterprise Bean Wizard - Entity Settings** dialog box (Entity EJBs only):
  - Specify a bean-managed or container-managed persistence, identify the primary key class and a primary key field name

Assembly and Deployment Guidelines (continued)

Adding an EJB to a J2EE application (continued):

- **New Enterprise Bean Wizard - Environment Entries** dialog box:
- **New Enterprise Bean Wizard - Enterprise Bean References** dialog box:
  - Declare all of the enterprise beans referenced by this enterprise bean. For each EJB reference, you enter the JNDI name used to access the EJB, the EJB type, and the names for the referenced EJB's home and remote interfaces; for example, `accountServer`, `entity`, `AccountHome`, `Account`.
- **New Enterprise Bean Wizard - Resource References** dialog box:
  - List any resource factories, such as `java.sql.DataSource`, referenced in the code of this enterprise bean
- **New Enterprise Bean Wizard - Security** dialog box:
  - Provide mapping information between security role references and security role names for the EJB
- **New Enterprise Bean Wizard - Transaction Management** dialog box:
  - Choose the transaction-management mechanism. Also select the transaction attribute for each of the EJB's methods if container-managed transactions are employed.
- **New Enterprise Bean Wizard - Review Settings** dialog box:
  - Displays the deployment descriptor generated for the EJB

Adding a Web component to a J2EE application:

- Choose File > New Web Component from the Deployment Tool Main Menu. The Help button displayed on each dialog box for the New Web Component Wizard invokes the help system with helpful information about the data elements contained on each screen. A brief explanation of each of the EJB Wizard's dialog boxes is given below. Consult the help documentation for more detailed information about specific screens and data elements.
- **New Web Component Wizard - WAR File General Properties** dialog box:
  - Select the location for the `.war` file, and enter the name used to display the `.war` file in the deployment tool browser
  - Select Contents > Add > Add Files to .WAR dialog box
    - Add Content Files: Use for HTML, JSP, GIF and similar types of files
    - Add Class Files: Use for bean or servlet class files
- **New Web Component Wizard - Choose Component Type** dialog box:
  - Choose from servlet, JSP, or No Web Component
- **New Web Component Wizard - Component General Properties** dialog box:
  - Identify the JSP file or servlet class, and provide its name, description, and icon
- **New Web Component Wizard - Component Initialization Parameters** dialog box:
  - List any initialization parameters that are referenced in the component's code and, optionally, provide default values to be used for these parameters
- **New Web Component Wizard - Component Aliases** dialog box:
  - List web component aliases. Calls to these aliases will be redirected to an instance of the web component.
- **New Web Component Wizard - Component Security** dialog box:
  - Names referenced in the component's `isUserInRole` methods, if any.

Assembly and Deployment Guidelines (continued)

Adding a Web component to a J2EE application (continued):

- **New Web Component Wizard - WAR File Environment** dialog box:
  - Environment entries referenced by the components in this `.war` file, as well as their Java types
- **New Web Component Wizard - Enterprise Bean References** dialog box:
  - Declare all of the enterprise beans referenced by this web component. For each EJB reference, you enter the JNDI name used to access the EJB, the EJB type, and the names for the referenced EJB's home and remote interfaces. For example, `bankmgrServer`, `session`, `BankMgrHome`, `BankMgr`.
- **New Web Component Wizard - Resource Factories** dialog box:
  - List any resource factories, such as `java.sql.DataSource`, referenced in the code of this web component
- **New Web Component Wizard - Context Parameters** dialog box:
  - Context parameters referenced by the components in the `.war` file
- **New Web Component Wizard - File References** dialog box:
  - The list of welcome files to be used by your `.war` file, as well as a listing of error codes and exception types to be trapped, and the corresponding URL to be called when this happens
- **New Web Component Wizard - Security** dialog box:
  - The authentication method and security constraints employed in the `.war` file

Deploying a J2EE application:

- Highlight the application in the Deployment Tool's Local Applications window
- Select Tools > Deploy Application.
- **Deploy *<AppName>* - Introduction** dialog box:
  - Choose the Target Server. You will most likely use the default local host setting for the course exercise.
  - Check the Return Client Jar check box, and review the jar file name and directory location. By default, the deployment tool places the client jar file in the same directory as the application ear file with the name *<AppName>*`Client.jar` where *<AppName>* is the application name.
  - Select Next.
- **Deploy *<AppName>* - JNDI Names** dialog box:
  - Enter the JNDI names for the application components.
  - Click Next, and then Finish to start the deployment process. Click OK when the application deployment has finished.

Undeploying a J2EE application:

- Highlight the application displayed in the Server Applications window.
- Click the Undeploy button.

Assembly and Deployment Guidelines (continued)

Updating J2EE application files:

- If the application is deployed, follow the instructions to undeploy it.
- Select the application in the Local Applications window.
- Choose Tools > Update Application Files from the Deployment Tool's menu.
- Verify the updated file list, and click OK.
- Highlight the application in the applications window, and choose File > Save to save the new configuration.

Viewing application deployment descriptors:

- Highlight the desired component in the Deployment Tool's Local Applications window.
- Select Tools > View Descriptor from the Deployment Tool's menu.
- Click the Save As button to copy the file for editing or distribution.

Banking Application Data Elements

The database used for this course contains three tables, a user table (J2EEAPPUSER), a customer table (J2EECUSTOMER), and an account table (J2EEACCOUNT). The user table contains user names, passwords, and associated user id codes. The user id code serves as the mechanism used to associate a user with the corresponding information stored in the customer and account tables. The user table contains the 34 users listed below.

username	password	userid
Alex	password	1
Anita	password	2
Bill	password	3
Boris	password	4
Bryan	password	5
Chris	password	6
Kenny	password	7
Dominic	password	8
Geronimo	password	9
Glynn	password	10
Gonzales	password	11
Gordon	password	12
Guido	password	13
Gwen	password	14
Hans	password	15
Hugo	password	16
Jeromy	password	17
Jim	password	18
Jo	password	19
Juan	password	20
Karin	password	21
KimLee	password	22
Maria	password	23
MaryAnn	password	24
Natasha	password	25
Pat	password	26
Patricia	password	27
Rashid	password	28
Scottie	password	29
Sinha	password	30
Sofia	password	31
Cartman	password	32
Yolonda	password	33
Al	password	34

Each user has a row in the customer table containing a user's first and last name, address, and phone number as well as a suite of rows in the account table, one row for each type of customer account; for example, the customer table contains the following information for user 34:

customerid	firstname	lastname	address	phone
34	Al	Davidson	Route 66, Chicago, IL	3021234567

User 34 also has three rows in the account table, each representing a separate type of account:

acctnumber	customerid	description	balance
34	34	CHECKING	1034.00
68	34	SAVINGS	10340.00
98	34	DISCOVER	1000.00

Use the appropriate database utility to view the complete list of data elements associated with each customer.

Important Terms and Concepts

Listed below is a summary of some of the classes, interfaces, and methods used during the development of the course lab application. Please refer to the JDK Javadoc for additional information on these and other classes, interfaces, and methods used in this course including parameters and return values.

`java.rmi`

`public class RemoteException extends IOException`

A `RemoteException` is the common superclass for a number of communication-related exceptions that might occur during the execution of a remote method call. Each method of a remote interface, an interface that extends `java.rmi.Remote`, must list `RemoteException` in its throws clause.

`java.sql`

`public interface Statement`

The object used for executing a static SQL statement and obtaining the results produced by it.

`public ResultSet executeQuery(String sql) throws SQLException`

Executes a SQL statement that returns a single `ResultSet`. Note that you must extract the columns from the result set in the order they are presented in the `SELECT` statement.

`public int executeUpdate(String sql) throws SQLException`

Executes an SQL `INSERT`, `UPDATE` or `DELETE` statement. Returns either the row count for `INSERT`, `UPDATE`, or `DELETE` or 0 for SQL statements that return nothing.

`public void close() throws SQLException`

Releases this `Statement` object's database and JDBC resources immediately instead of waiting for this to happen when it is automatically closed. When a `Statement` is closed, its current `ResultSet`, if one exists, is also closed.

`public interface PreparedStatement extends Statement`

An object that represents a precompiled SQL statement. A SQL statement is precompiled and stored in a `PreparedStatement` object. This object can then be used to efficiently execute this statement multiple times.

`public void setObject(int parameterIndex, Object x)`
`throws SQLException`

Sets the value of the designated parameter using the given object. The second parameter must be of type `Object`; therefore, the `java.lang` equivalent objects should be used for built-in types. The JDBC specification specifies a standard mapping from Java `Object` types to SQL types. The given argument will be converted to the corresponding SQL type before being sent to the database.

`public void setString(int parameterIndex, String x)`
`throws SQLException`

Sets the designated parameter to a Java `String` value. The driver converts this to an SQL `VARCHAR` or `LONGVARCHAR` value (depending on the argument's size relative to the driver's limits on `VARCHAR` values) when it sends it to the database.

`public boolean execute() throws SQLException`

Executes any kind of SQL statement. Some prepared statements return multiple results; the `execute` method handles these complex statements as well as the simpler form of statements handled by the methods `executeQuery` and `executeUpdate`.

`public ResultSet executeQuery() throws SQLException`

Executes the SQL query in this `PreparedStatement` object, and returns the result set generated by the query.

`public int executeUpdate() throws SQLException`

Executes the SQL `INSERT`, `UPDATE`, or `DELETE` statement in this `PreparedStatement` object. In addition, SQL statements that return nothing, such as SQL DDL statements, can be executed.

`public interface ResultSet`

A `ResultSet` provides access to a table of data. A `ResultSet` object is usually generated by executing a `Statement`. A `ResultSet` maintains a cursor pointing to its current row of data. Initially the cursor is positioned before the first row. The `next` method moves the cursor to the next row.

`public boolean next() throws SQLException`

Moves the cursor down one row from its current position. A `ResultSet` cursor is initially positioned before the first row; the first call to next makes the first row the current row; the second call makes the second row the current row, and so on.

`public void close() throws SQLException`

Releases this `ResultSet` object's database and JDBC resources immediately instead of waiting for this to happen when it is automatically closed. A `ResultSet` is automatically closed by the `Statement` that generated it when that `Statement` is closed, re-executed, or is used to retrieve the next result from a sequence of multiple results.

`double getDouble(int columnIndex) or`
`double getDouble(String columnName)`

Returns `double` value in specified column of current record.

`int getInt(int columnIndex) or int getInt(String columnName)`

Gets the value of a column in the current row as a Java `int`.

`String getString(int columnIndex) or`
`String getString(String columnName)`

Gets the value of a column in the current row as a Java `String`.

`public interface Connection`

A connection (session) with a specific database. Within the context of a `Connection`, SQL statements are executed and results are returned. Note: By default, the `Connection` automatically commits changes after executing each statement. If auto commit has been disabled, an explicit commit must be perfromed or database changes will not be saved.

`public Statement createStatement() throws SQLException`

Creates a `Statement` object for sending SQL statements to the database. SQL statements without parameters are normally executed using `Statement` objects. If the same SQL statement is executed many times, it is more efficient to use a `PreparedStatement`. JDBC 2.0 Result sets created using the returned `Statement` will have forward-only type, and read-only concurrency, by default.

`public PreparedStatement prepareStatement(String sql)`
`throws SQLException`

Creates a `PreparedStatement` object for sending parameterized SQL statements to the database. A SQL statement with or without IN parameters can be pre-compiled and stored in a `PreparedStatement` object. This object can then be used to efficiently execute this statement multiple times.

`public void close() throws SQLException`

Releases the connection if it is no longer needed by the current transaction, and returns it to the pool if there is one. If there is no pool, the connection is freed.

java.util

public class HashMap extends AbstractMap
implements Map, Cloneable, Serializable

Hash table based implementation of the Map interface. This implementation provides all of the optional map operations and permits null values and the null key.

Object clone()

Returns a shallow copy of this HashMap instance: the keys and values themselves are not cloned.

Object put(Object key, Object value)

Associates the specified value with the specified key in this map.

Object get(Object key)

Returns the value to which this map maps the specified key.

Set keySet()

Returns a set view of the keys contained in this map.

Object remove(Object key)

Removes the mapping for this key from this map, if present.

public interface Collection

The root interface in the collection hierarchy. A collection represents a group of objects, known as its elements. Some collections allow duplicate elements and others do not. Some are ordered and others unordered. The SDK does not provide any direct implementations of this interface: It provides implementations of more specific subinterfaces like Set and List. This interface is typically used to pass collections around and manipulate them where maximum generality is desired.

public boolean add(Object o)

Ensures that this collection contains the specified element (optional operation). Returns true if this collection changed as a result of the call. (Returns false if this collection does not permit duplicates and already contains the specified element.)

public boolean isEmpty()

Returns true if this collection contains no elements.

public Iterator iterator()

Returns an iterator over the elements in this collection. There are no guarantees concerning the order in which the elements are returned (unless this collection is an instance of some class that provides a guarantee).

public class ArrayList extends AbstractList
implements List, Cloneable, Serializable

Resizable-array implementation of the List interface. Implements all optional list operations, and permits all elements, including null. In addition to implementing the List interface, this class provides methods to manipulate the size of the array that is used internally to store the list. (This class is roughly equivalent to Vector, except that it is unsynchronized.)

public boolean add(Object o)

Appends the specified element to the end of this list.

public interface Set extends Collection

A collection that contains no duplicate elements. More formally, sets contain no pair of elements e1 and e2 such that e1.equals(e2), and at most one null element. As implied by its name, this interface models the mathematical set abstraction.

Iterator iterator()

Returns an iterator over the elements in this set.

public class Vector extends AbstractList
implements List, Cloneable, Serializable

The `Vector` class implements a growable array of objects. Like an array, it contains components that can be accessed using an integer index. However, the size of a `Vector` can grow or shrink as needed to accommodate adding and removing items after the `Vector` has been created.

`public void addElement(Object obj)`
Adds the specified component to the end of this vector, increasing its size by one. The capacity of this vector is increased if its size becomes greater than its capacity.

`public interface Iterator`
An iterator over a collection. `Iterator` takes the place of `Enumeration` in the Java collections framework.

`public boolean hasNext()`
Returns `true` if the iteration has more elements. (In other words, returns `true` if next would return an element rather than throwing an exception.)

`javax.ejb`

`public class CreateException extends java.lang.Exception`
The `CreateException` exception must be included in the throws clauses of all creator methods defined in an enterprise Bean's remote interface. The exception is used as a standard application-level exception to report a failure to create an entity EJB object.

`public class EJBException extends java.lang.RuntimeException`
The `EJBException` exception is thrown by an enterprise Bean instance to its container to report that the invoked business method or callback method could not be completed because of an unexpected error (for example, the instance failed to open a database connection).

`public class FinderException extends java.lang.Exception`
The `FinderException` exception must be included in the throws clause of every finder method of an entity bean's home interface. The exception is used as a standard application-level exception to report a failure to find the requested EJB object.

`public class ObjectNotFoundException extends FinderException`
The `ObjectNotFoundException` exception is thrown by a finder method to indicate that the specified EJB object does not exist. Only the finder methods that are declared to return a single EJB object use this exception. This exception should not be thrown by finder methods that return a collection of EJB objects (they should return a null collection, instead).

`public class RemoveException extends java.lang.Exception`
The `RemoveException` exception is thrown at an attempt to remove an EJB object when the enterprise bean or the container does not allow the EJB object to be removed.

`public interface EntityContext extends EJBContext`
The `EntityContext` interface provides an instance with access to the container-provided runtime context of an entity enterprise Bean instance. The container passes the `EntityContext` interface to an entity enterprise bean instance after the instance has been created. The `EntityContext` interface remains associated with the instance for the lifetime of the instance. Note that the information that the instance obtains using the `EntityContext` interface (such as the result of the `getPrimaryKey` method) might change, as the container assigns the instance to different EJB objects during the instance's life cycle.

`java.lang.Object getPrimaryKey()`
Obtain the primary key of the EJB object that is currently associated with this instance.

`public interface SessionBean extends EnterpriseBean`

The `SessionBean` interface is implemented by every session enterprise bean class. The container uses the `SessionBean` methods to notify the enterprise bean instances of the instance's life cycle events.

`void ejbActivate()`

The `activate` method is called when the instance is activated from its "passive" state.

`void ejbPassivate()`

The `passivate` method is called before the instance enters the "passive" state.

`void ejbRemove()`

A container invokes this method before it ends the life of the session object.

`void setSessionContext(SessionContext ctx)`

Set the associated session context.

`public interface SessionContext extends EJBContext`

The `SessionContext` interface provides access to the runtime session context that the container provides for a session enterprise bean instance. The container passes the `SessionContext` interface to an instance after the instance has been created. The session context remains associated with the instance for the lifetime of the instance.

`EJBObject getEJBObject()`

Obtain a reference to the EJB object that is currently associated with the instance.

`javax.naming`

`public interface Context`

This interface represents a naming context, which consists of a set of name-to-object bindings. It contains methods for examining and updating these bindings.

`public class InitialContext extends java.lang.Object implements Context`

This class is the starting context for performing naming operations. All naming operations are relative to a context. The initial context implements the `Context` interface and provides the starting point for resolution of names.

`java.lang.Object lookup(String name)`

Retrieves the named object. If name is empty, returns a new instance of this context (which represents the same naming context as this context, but its environment might be modified independently and it may be accessed concurrently).

`javax.rmi`

`public class PortableRemoteObject extends Object`

Server implementation objects can either inherit from `javax.rmi.PortableRemoteObject`, or they can implement a remote interface and then use the `exportObject` method to register themselves as a server object.

`public static Object narrow(Object narrowFrom, Class narrowTo) throws ClassCastException`

Makes a `Remote` object ready for remote communication. This normally happens implicitly when the object is sent or received as an argument on a remote method call, but, in some circumstances, it is useful to perform this action by making an explicit call.

javax.servlet

public interface Servlet

Defines methods that all servlets must implement. A servlet is a small Java program that runs within a Web server. Servlets receive and respond to requests from Web clients, usually across HTTP, the Hypertext Transfer Protocol. To implement this interface, you can write a generic servlet that extends `javax.servlet.GenericServlet` or an HTTP servlet that extends `javax.servlet.http.HttpServlet`.

public ServletConfig getServletConfig()

Returns a `ServletConfig` object, which contains initialization and startup parameters for this servlet. The `ServletConfig` object returned is the one passed to the `init` method.

public java.lang.String getServletInfo()

Returns information about the servlet, such as author, version, and copyright.

public void init(ServletConfig config) throws ServletException

Called by the servlet container to indicate to a servlet that the servlet is being placed into service. The servlet container calls the `init` method exactly once after instantiating the servlet. The `init` method must complete successfully before the servlet can receive any requests.

public interface ServletConfig

A servlet configuration object used by a servlet container used to pass information to a servlet during initialization.

public ServletContext getServletContext()

Returns a reference to the `ServletContext` in which the servlet is executing.

public interface ServletContext

Defines a set of methods that a servlet uses to communicate with its servlet container, for example, to get the MIME type of a file, dispatch requests, or write to a log file. There is one context per "web application" per Java Virtual Machine. (A "web application" is a collection of servlets and content installed under a specific subset of the server's URL namespace such as /catalog and possibly installed using a `.war` file.)

public RequestDispatcher getRequestDispatcher(String path)

Returns a `RequestDispatcher` object that acts as a wrapper for the resource located at the given path. A `RequestDispatcher` object can be used to forward a request to the resource or to include the resource in a response. The resource can be dynamic or static.

public abstract class ServletOutputStream

Provides an output stream for sending binary data to the client. A `ServletOutputStream` object is normally retrieved using the `ServletResponse.getOutputStream` method.

extends java.io.OutputStreampublic interface ServletRequest

Defines an object to provide client request information to a servlet. The servlet container creates a `ServletRequest` object and passes it as an argument to the servlet's `service` method. A `ServletRequest` object provides data including parameter name and values, attributes, and an input stream.

public java.lang.Object getAttribute(String name)

Returns the value of the named attribute as an `Object` or `null` if no attribute of the given name exists.

public void setAttribute(String name, Object o)

Stores an attribute in this request. Attributes are reset between requests. This method is most often used with `RequestDispatcher`.

public java.lang.String getParameter(String name)

Returns the value of a request parameter as a `String` or `null` if the parameter does not exist. Request parameters are extra information sent with the request. For HTTP servlets, parameters are contained in the query string or posted form data.

`public interface ServletResponse`

Defines an object to assist a servlet in sending a response to the client. The servlet container creates a `ServletResponse` object and passes it as an argument to the servlet's `service` method.

public java.io.PrintWriter getWriter() throws java.io.IOException

Returns a `PrintWriter` object that can send character text to the client. The character encoding used is the one specified in the `charset=` property of the `setContentType(String)` method, which must be called before calling this method for the `charset` to take effect.

public ServletOutputStream getOutputStream()
throws java.io.IOException

Returns a `ServletOutputStream` suitable for writing binary data in the response. The servlet cntainer does not encode the binary data. Either this method or `getWriter` may be called to write the body, not both.

public void setContentType(String type)

Sets the content type of the response being sent to the client. The content type may include the type of character encoding used, for example, `text/html; charset=ISO-8859-4`. If obtaining a `PrintWriter`, this method should be called first.

`public interface RequestDispatcher`

Defines an object that receives requests from the client and sends them to any resource (such as a servlet, HTML file, or JSP file) on the server. The servlet container creates the `RequestDispatcher` object, which is used as a wrapper around a server resource located at a particular path or given by a particular name.

public void forward(ServletRequest request,
ServletResponse response)
throws ServletException, java.io.IOException

Forwards a request from a servlet to another resource (servlet, JSP file, or HTML file) on the server. This method allows one servlet to do preliminary processing of a request and another resource to generate the response. For a `RequestDispatcher` obtained using `getRequestDispatcher`, the `ServletRequest` object has its path elements and parameters adjusted to match the path of the target resource. Forward should be called before the response has been committed to the client (before response body output has been flushed). If the response already has been committed, this method throws an `IllegalStateException`. Uncommitted output in the response buffer is automatically cleared before the forward.

`javax.servlet.http`

`public interface HttpServletRequest extends ServletRequest`

Extends the `ServletRequest` interface to provide request information for HTTP servlets.

public HttpSession getSession(boolean create)

Returns the current `HttpSession` associated with this request or, if there is no current session and `create` is `true`, returns a new session. If `create` is `false` and the request has no valid `HttpSession`, this method returns `null`.

`public interface HttpServletResponse extends ServletResponse`

Extends the `ServletResponse` interface to provide HTTP-specific functionality in sending a response. For example, it has methods to access HTTP headers and cookies.

`public interface HttpSession`

Provides a way to identify a user across more than one page request or visit to a Web site and to store information about that user.

`public java.lang.Object getAttribute(String name)`

Returns the object bound with the specified `name` in this session or `null` if no object is bound under the `name`.

`public void invalidate()`

Invalidates this session and unbinds any objects bound to it.

`public boolean isNew()`

Returns `true` if the client does not yet know about the session or if the client chooses not to join the session. For example, if the server used only cookie-based sessions, and the client had disabled the use of cookies, then a session would be new on each request.

`public void setAttribute(String name, Object value)`

Binds an object to this session, using the `name` specified. If an object of the same name is already bound to the session, the object is replaced.

`javax.sql`

`public interface DataSource`

A `DataSource` object is a factory for `Connection` objects. An object that implements the `DataSource` interface will typically be registered with a JNDI service provider. A JDBC driver that is accessed using the `DataSource` API does not automatically register itself with the `DriverManager`.

`java.sql.Connection getConnection() throws java.sql.SQLException`

Attempts to establish a database connection.

Filename:	appendix.doc
Directory:	H:\FJ310_revC_0301\BOOK\EXERCISE\Exercise_Originals
Template:	C:\Program Files\Microsoft Office\Templates\Normal.dot
Title:	Case Study: The New User Interface
Subject:	
Author:	jdibble
Keywords:	
Comments:	
Creation Date:	10/16/00 8:39 AM
Change Number:	62
Last Saved On:	03/19/01 9:42 AM
Last Saved By:	randy
Total Editing Time:	325 Minutes
Last Printed On:	04/24/01 12:42 PM

As of Last Complete Printing

Number of Pages:	29
Number of Words:	3,203 (approx.)
Number of Characters:	18,262 (approx.)