CHAPTER 5

Building Java Applications with Ant

This chapter explains the techniques involved in using Ant to build and deploy Java applications in an orchestrated fashion. Once you understand these fundamental techniques, you will be ready for the complexities of Chapter 6, "Building J2EE Applications with Ant," in which we build a Model 2 application complete with EJBs, servlets, and JSPs.

During the course of this chapter, we will build a "Hello World" example. Because the emphasis is on how to *build* components with Ant—and not the mechanics of implementing these various components—the example is meant to be as simple as possible. We will package the application and the common JAR, construct a buildfile that creates an applet, and construct a master buildfile that coordinates the entire build.

The source code for the Hello World example is divided into several directories. Each directory has its own Ant buildfile, which contains instructions to compile the components and package the binaries. Each directory also includes the requisite configuration files (such as manifest files). The master Ant buildfile, located in the root directory, calls the other buildfiles and coordinates the entire build.

This divide-and-conquer technique for organizing components into separate directories is quite common with Ant. It may seem like overkill on a simple project like this one, but consider building a system with 50 components. Each component has its own set of deployment descriptors and configuration files, and each component is deployed in different containers. The divide-and-conquer technique becomes necessary to mange the complexity—it also makes it easier to reuse components.

Following is the directory structure for the Hello World project. We use this same structure for the more complex example in Chapter 6:

```
Model 2 Hello World root
+—-Model
+—-Application
+—-Applet
+—-WebApplication
+—-EJBeans
```

The Model directory holds the common code (in this simple project, only the application will access the common code). The Application directory holds the Java application code, including the manifest file that marks the deployment JAR as executable. The Applet directory holds the applet code. The WebApplication and EJBeans directories are discussed in Chapter 6.

Hello World Model Project

In Chapter 4 we used a Hello World example that was contained in one file. The Hello World example in this chapter is more complex because it uses several classes.

This section explains the basic structure for all the buildfiles and directories you use in this example and the rest of the book. We introduce the three Java class files: a GreetingBean class, a GreetingFactory class, and a Greeting interface. Then, we discuss how to build these files with Ant and break down the Ant buildfiles' target execution step by step. We also explain how to use the Ant command-line utility to build the files with Ant.

Overview of Model Classes

The GreetingFactory knows how to create a Greeting object. Here is the listing for the GreetingFactory:

```
package xptoolkit.model;

public class GreetingFactory {
    private GreetingFactory(){}

    public Greeting getGreeting()throws Exception {
     String clazz = System.getProperty("Greeting.class",
                            "xptoolkit.model.GreetingBean");
```

```
            return (Greeting)Class.forName(clazz).newInstance();
        }

        public static GreetingFactory getGreetingFactory(){
            return new GreetingFactory();
        }

    }
```

Next we have a Greeting interface that defines the contract of a Greeting object—that is, what type of behavior it supports. The Greeting interface is as follows:

```
    package xptoolkit.model;

    public interface Greeting extends java.io.Serializable{
        public String getGreeting();
    }
```

Finally, the GreetingBean class implements the Greeting interface. GreetingBean is defined as follows:

```
    package xptoolkit.model;

    public class GreetingBean implements Greeting{

        public GreetingBean(){}

        public String getGreeting(){
          return "Hello World!";
        }
    }
```

The GreetingBean returns the message "Hello World!" just like the message in the Chapter 4 application. To create a Greeting instance, you use the GreetingFactory. The default implementation of the GreetingFactory gets the implementation class from a property and instantiates an instance of that class with the Class.forName().newInstance() method. It casts the created instance to the Greeting interface.

These two lines of code create the Greeting instance from the GreetingFactory's getGreeting() method:

```
           String clazz = System.getProperty("Greeting.class",
                                 "xptoolkit.model.GreetingBean");
          return (Greeting)Class.forName(clazz).newInstance();
```

Thus any class that implements the Greeting interface can be substituted as the Greeting.class system property. Then, when the class is instantiated with the factory's getGreeting() method, the application uses the new implementation of the Greeting interface.

We use this technique in Chapter 6 to add support for EJBs to the Web application seamlessly. We create an Ant buildfile that can deploy the same Web application to use either enterprise beans or another bean implementation just by setting an Ant property. Later, we also map the Greeting interface with the use bean action of a JSP when we implement the Model 2 using servlets and JSP.

Creating a Project Directory Structure for Model

This part of the sample application uses the smallest buildfile. Basically, we just need to create a JAR file that acts as a common library. We don't need any special manifest file or deployment files. This is the most basic buildfile and directory structure you will see in this example. Here is the directory structure for the Model directory:

```
Root of Model
|   build.xml
|
+--src
    +--xptoolkit
        \--model
                GreetingFactory.java
                Greeting.java
                GreetingBean.java
```

Notice that there are only four files in the Model directory and subdirectories. Also notice that the name of the Ant file is build.xml. Remember from Chapter 4 that build.xml is the default buildfile; if Ant is run in this directory, it automatically finds build.xml without you having to specify it on the command line. Let's dig into the model buildfile.

Creating a Buildfile for a Shared Library

The model buildfile has six targets: setProps, init, clean, delete, prepare, compile, package, and all. The buildfiles in this example have similar targets:

- setProps sets up the output directory ("outputdir") property if it is not already set. This behavior is important so we can easily set a different output directory from the command line or from another buildfile that invokes this buildfile, and yet have a reasonable default.
- init initializes all the other properties relative to the "outputdir" property defined in the setProps target; init depends on setProps.
- clean cleans up the output directories and the output JAR file.

- prepare creates the output directories if they do not already exist.
- compile compiles the Java source files for the model into the build directory defined in the init target.
- package packages the compiled Java source into a JAR file.
- all runs all the tags. It is the default target of this build project.

Analysis of the Model Project Buildfile

Listing 5.1 shows the entire buildfile for the model project. In this section we provide a step-by-step analysis of how this buildfile executes. All the buildfiles in the Hello World example are structured in a similar fashion, so understanding the model project's buildfile is essential to understanding the others. A quick note on naming conventions: As you see from the first line of code in Listing 5.1, the project name for this buildfile is "model". Thus we refer to this buildfile as the *model project buildfile*. This naming convention becomes essential once we begin dealing with the five other buildfiles in this project.

```
<project name="model" default="all" >


    <target name="setProps" unless="setProps"
                          description="setup the properties.">
        <property name="outdir" value="/tmp/app/" />
    </target>

    <target name="init" depends="setProps"
                          description="initialize the properties.">
        <tstamp/>
        <property name="local_outdir" value="${outdir}/model" />
        <property name="build" value="${local_outdir}/classes" />
        <property name="lib" value="${outdir}/lib" />
        <property name="model_jar" value="${lib}/greetmodel.jar" />
    </target>

    <target name="clean" depends="init"
                    description="clean up the output directories and jar.">
        <delete dir="${local_outdir}" />
        <delete file="${model_jar}" />
    </target>
```

Listing 5.1 The Hello World model project buildfile. (continues)

```
    <target name="prepare" depends="init"
                          description="prepare the output directory.">
        <mkdir dir="${build}" />
        <mkdir dir="${lib}" />
    </target>

    <target name="compile" depends="prepare"
                          description="compile the Java source.">
        <javac srcdir="./src" destdir="${build}" />
    </target>

    <target name="package" depends="compile"
                   description="package the Java classes into a jar.">
        <jar jarfile="${model_jar}"
           basedir="${build}" />
    </target>

    <target name="all" depends="clean,package"
                   description="perform all targets."/>

</project>
```

Listing 5.1 The Hello World model project buildfile.

Let's go over the model project buildfile and each of its targets in the order they execute. First, the model project sets the all target as the default target, as follows:

```
<project name="model" default="all" >
```

The all target is executed by default, unless we specify another target as a command-line argument of Ant. The all target depends on the clean and package targets. The clean target depends on the init target. The init target depends on the setProps target, and thus the setProps target is executed first.

Following is the setProps target defined in build.xml:

```
<target name="setProps" unless="setProps"
                      description="setup the properties.">
    <property name="outdir" value="/tmp/app/" />
</target>
```

The setProps target executes only if the "setProps" property is not set (unless="setProps"). Thus, if a parent buildfile calls this buildfile, it can set the "setProps" property and override the value of outdir so that the setProps target of this file does not execute (we give an example of this later). If the setProps target executes, it sets the value of outdir to /tmp/app.

Next, the init target is executed. Following is the init target defined in build.xml:

```
<target name="init" depends="setProps"
                      description="initialize the properties.">
    <tstamp/>

    <property name="local_outdir" value="${outdir}/model" />
    <property name="build" value="${local_outdir}/classes" />
    <property name="lib" value="${outdir}/lib" />
    <property name="model_jar" value="${lib}/greetmodel.jar" />
</target>
```

The init target uses the tstamp task to get the current time, which is used by the javac task to see if a source file is out of data and needs to be compiled. The init target defines several properties that refer to directories and files needed to compile and deploy the model project. We will discuss the meaning of these properties because all the other buildfiles for this example use the same or similar properties. The init target defines the following properties:

- The “local_outdir” property defines the output directory of all the model project’s intermediate files (Java class files).
- The “build” property defines the output directory of the Java class files.
- The “lib” property defines the directory that holds the common code libraries (JAR files) used for the whole Model 2 Hello World example application.
- The “model_jar” property defines the output JAR file for this project.

Using Literals

As a general rule, if you use the same literal twice, you should go ahead and define it in the init target. You don’t know how many times we’ve shot ourselves in the foot by not following this rule. This buildfile is fairly simple, but the later ones are more complex. Please learn from our mistakes (and missing toes).

Now that all the clean target’s dependencies have executed, the clean target can execute. The clean target deletes the intermediate files created by the compile and the output common JAR file, which is the output of this project. Here is the code for the clean target:

```
<target name="clean" depends="init"
         description="clean up the output directories and jar.">
```

```
    <delete dir="${local_outdir}" />
    <delete file="${model_jar}" />

  </target>
```

Remember that the all target depends on the clean and package targets. The clean branch and all its dependencies have now executed, so it is time to execute the package target branch (a *branch* is a target and all its dependencies). The package target depends on the compile target, the compile target depends on the prepare target, and the prepare target depends on the init target, which has already been executed.

Thus, the next target that executes is prepare, because all its dependencies have already executed. The prepare target creates the build output directory, which ensures that the lib directory is created. The prepare target is defined as follows:

```
<target name="prepare" depends="init"
                      description="prepare the output directory.">

    <mkdir dir="${build}" />
    <mkdir dir="${lib}" />
</target>
```

The next target in the package target branch that executes is the compile target—another dependency of the package target. The compile target compiles the code in the src directory to the build directory, which was defined by the "build" property in the init target. The compile target is defined as follows:

```
 <target name="compile" depends="prepare"
                      description="compile the Java source.">

    <javac srcdir="./src" destdir="${build}"/>
 </target>
```

Now that all the target dependencies of the package target have been executed, we can run the package target. Whew! The package target packages the Java classes created in the compile target into a JAR file that is created in the common lib directory. The package target is defined as follows:

```
<target name="package" depends="compile"
               description="package the Java classes into a jar.">

    <jar jarfile="${model_jar}"
       basedir="${build}" />
</target>
```

Running an Ant Buildfile

In this section, we discuss how to run the Hello World model project buildfile. There are three steps to running the Ant buildfile:

1. Set up the environment.
2. Go to the directory that contains the build.xml file for the model.
3. Run Ant.

Successfully running the buildscript gives us the following output:

```
Buildfile: build.xml

setProps:

init:

clean:

prepare:
    [mkdir] Created dir: C:\tmp\app\model\classes

compile:
    [javac] Compiling 3 source files to C:\tmp\app\model\classes

package:
      [jar] Building jar: C:\tmp\app\lib\greetmodel.jar

all:

BUILD SUCCESSFUL

Total time: 3 seconds
```

If you do not get this output, check that the properties defined in the init target make sense for your environment. If you are on a Unix platform and the buildfile is not working, make sure that the /tmp directory exists and that you have the rights to access it. Alternatively, you could run the previous script by doing the following on the command line:

```
$ ant -DsetProps=true -Doutdir=/usr/rick/tmp/app
```

Basically, you want to output to a directory that you have access to, just in case you are not the administrator of your own box. If from some reason Ant still does not run, make sure you set up the Ant environment variables (refer to Chapter 4 for details).

After successfully running Ant, the output directory for the model project will look like this:

```
Root of output directory
\--app
    +--lib
    |       greetmodel.jar
    |
    \--model
        \--classes
            \--xptoolkit
                \--model
                        GreetingFactory.class
                        Greeting.class
                        GreetingBean.class
```

Notice that all the intermediate files to build the JAR file are in the model subdirectory. The output from this project is the greetmodel.jar file, which is in ${outdir}/app/lib. The next project, the application project, needs this JAR file in order to compile. In the next section, we discuss how to build a standalone Java application with Ant that uses the JAR file (greetmodel.jar) from the model project.

Hello World Application Project

The goal of the Hello World application project is to create a standalone Java application that uses greetmodel.jar to get the greeting message. The application project buildfile is nearly identical to the model project buildfile, so we focus our discussion on the differences between the two buildfiles. We also explain how to make a JAR file an executable JAR file.

Overview of Application Java Classes

The Java source code for this application is as simple as it gets for the Hello World Model 2 examples. Here is the Java application:

```
package xptoolkit;

import xptoolkit.model.GreetingFactory;
import xptoolkit.model.Greeting;

public class HelloWorld{

    public static void main(String []args)throws Exception{
       Greeting greet = (Greeting)
```

```
             GreetingFactory.getGreetingFactory().getGreeting();

           System.out.println(greet.getGreeting());

        }
    }
```

As you can see, this application imports the GreetingFactory class and the Greeting interface from the model project. It uses the GreetingFactory to get an instance of the Greeting interface, and then uses the instance of the Greeting interface to print the greeting to the console.

Creating a Project Directory Structure for the Application

The directory structure of the Hello World Java application is as follows:

```
Hello World Application root
|   build.xml
|
+--src
|   |
|   +--xptoolkit
|           HelloWorld.java
|
\--META-INF
        MANIFEST.MF
```

Notice the addition of the META-INF directory, which holds the name of the manifest file we will use to make the application's JAR file executable. The only other file that this project needs is not shown; the file is greetmodel.jar, which is created by the model project (the reason for this will become obvious in the following sections).

Creating a Manifest File for a Standalone Application

The goal of this application is for it to work as a standalone JAR file. To do this, we need to modify the manifest file that the application JAR file uses to include the main class and the dependency on greetmodel.jar. The manifest entries that this application needs look something like this:

```
Manifest-Version: 1.0
Created-By: Rick Hightower
Main-Class: xptoolkit.HelloWorld
Class-Path: greetmodel.jar
```

The Class-Path manifest entry specifies the JAR files that the JAR file that holds the Hello World Java application needs to run (in our case, greetmodel.jar). The Main-Class manifest entry specifies the main class of the JAR file—that is, the class with the main method that is run when the executable JAR file executes.

Creating an Ant Buildfile for a Standalone Application

Listing 5.2 shows the application project buildfile; you'll notice that it is very similar to the model project buildfile. It is divided into the same targets as the model project buildfile: setProps, init, clean, delete, prepare, mkdir, compile, package, and all. The application project buildfile defines the properties differently, but even the property names are almost identical (compare with the model project buildfile in Listing 5.1).

```
<project name="application" default="all" >


    <target name="setProps" unless="setProps"
                          description="setup the properties.">
<property name="outdir" value="/tmp/app" />
    </target>

    <target name="init" depends="setProps"
                          description="initialize the properties.">
      <tstamp/>
      <property name="local_outdir" value="${outdir}/java_app" />
      <property name="build" value="${local_outdir}/classes" />
      <property name="lib" value="${outdir}/lib" />
      <property name="app_jar" value="${lib}/greetapp.jar" />
    </target>

    <target name="clean" depends="init"
                          description="clean up the output directories.">
         <delete dir="${build}" />
         <delete file="${app_jar}" />
    </target>

    <target name="prepare" depends="init"
                          description="prepare the output directory.">
        <mkdir dir="${build}" />
        <mkdir dir="${lib}" />
    </target>
```

Listing 5.2 Hello World application project buildfile. (continues)

06 ch05.qxd 10/10/2001 9:38 AM Page 103

```
    <target name="compile" depends="prepare"
                         description="compile the Java source.">

        <javac srcdir="./src" destdir="${build}">
            <classpath >

                <fileset dir="${lib}">
                    <include name="**/*.jar"/>
                </fileset>

            </classpath>

        </javac>

    </target>

    <target name="package" depends="compile"
                  description="package the Java classes into a jar.">
        <jar jarfile="${app_jar}"
             manifest="./META-INF/MANIFEST.MF"
           basedir="${build}" />
    </target>

    <target name="all" depends="clean,package"
                         description="perform all targets."/>

</project>
```

Listing 5.2 Hello World application project buildfile.

One of the differences in the application project buildfile is the way that it compiles the Java source:

```
    <target name="compile" depends="prepare"
                         description="compile the Java source.">

        <javac srcdir="./src" destdir="${build}">
            <classpath >

                <fileset dir="${lib}">
                    <include name="**/*.jar"/>
                </fileset>

            </classpath>

        </javac>

    </target>
```

Notice that the compile target specifies all the JAR files in the common lib directory (<include name="**/*.jar"/>). The greetmodel.jar file is in the common lib directory, so it is included when the javac task compiles the source. Another difference is the way the application project's buildfile packages the Ant source as follows:

```
<target name="package" depends="compile"
               description="package the Java classes into a jar.">
    <jar jarfile="${app_jar}"
         manifest="./META-INF/MANIFEST.MF"
       basedir="${build}" />
</target>
```

Notice that the package target uses the jar task as before, but the jar task's manifest is set to the manifest file described earlier. This is unlike the model project buildfile, which did not specify a manifest file; the model used the default manifest file. The application project buildfile's manifest file has the entries that allow us to execute the JAR file from the command line.

In order to run the Hello World Java application, after we run the application project's buildfile, we go to the output common lib directory (tmp/app/lib) and run Java from the command line with the -jar command-line argument, as follows:

```
$ java -jar greetapp.jar
Hello World!
```

You may wonder how it loaded the Greeting interface and GreetingFactory class. This is possible because the manifest entry Class-Path causes the JVM to search for any directory or JAR file that is specified (refer to the JAR file specification included with the Java Platform documentation for more detail). The list of items (directory or JAR files) specified on the Class-Path manifest entry is a relative URI list. Because the greetmodel.jar file is in the same directory (such as /tmp/app/lib) and it is specified on the Class-Path manifest, the JVM finds the classes in greetmodel.jar.

One issue with the application project is its dependence on the model project. The model project must be executed before the application project. How can we manage this? The next section proposes one way to manage the situation with an Ant buildfile.

Hello World Main Project

The Hello World Java application depends on the existence of the Hello World model common library file. If we try to compile the application before the model, we get an error. The application requires the model, so we need a way to call the model project buildfile and the application project buildfile in the right order.

Creating a Master Buildfile

We can control the execution of two buildfiles by using a master buildfile. The master buildfile shown in Listing 5.3 is located in the root directory of the Model 2 Hello World Example of the main project. This buildfile treats the model and application buildfile as subprojects (the model and application projects are the first of many subprojects that we want to fit into a larger project).

```
<project name="main" default="build" >


    <target name="setProps" unless="setProps"
                          description="setup the properties.">
         <property name="outdir" value="/tmp/app" />
         <property name="setProps" value="true" />
    </target>


    <target name="init" depends="setProps"
                          description="initialize the properties.">
          <property name="lib" value="${outdir}/lib" />
    </target>

    <target name="clean" depends="init"
                          description="clean up the output directories.">
        <ant dir="./Model" target="clean">
            <property name="outdir" value="${outdir}" />
            <property name="setProps" value="true" />

        </ant>

        <ant dir="./Application" target="clean">
            <property name="outdir" value="${outdir}" />
            <property name="setProps" value="true" />
        </ant>

        <delete dir="${outdir}" />

    </target>
```

Listing 5.3 Hello World master buildfile. (continues)

06 ch05.qxd 10/10/2001 9:38 AM Page 106

```
        <target name="prepare" depends="init"
                            description="prepare the output directory.">
        <mkdir dir="${build}" />
        <mkdir dir="${lib}" />
    </target>


    <target name="build" depends="prepare"
            description="build the model and application modules.">

        <ant dir="./model" target="package">
            <property name="outdir" value="${outdir}" />
            <property name="setProps" value="true" />
        </ant>

        <ant dir="./application" target="package">
            <property name="outdir" value="${outdir}" />
            <property name="setProps" value="true" />
        </ant>
    </target>

</project>
```

Listing 5.3 Hello World master buildfile.

Analysis of the Master Buildfile

Notice that the main project buildfile simply delegates to the application and model subproject and ensures that the subprojects' buildfiles are called in the correct order. For example, when the clean target is executed, the main project's buildfile uses the ant task to call the model project's clean target. Then, the main project calls the application project's clean target using the ant task again. Both are demonstrated as follows:

```
<target name="clean" depends="init"
                      description="clean up the output directories.">
    <ant dir="./Model" target="clean">
        <property name="outdir" value="${outdir}" />
        <property name="setProps" value="true" />
    </ant>

    <ant dir="./Application" target="clean">
        <property name="outdir" value="${outdir}" />
        <property name="setProps" value="true" />
    </ant>
```

```
        <delete dir="${outdir}" />

    </target>
```

A similar strategy is used with the main project's build target. The build target calls the package target on both the model and application subprojects, as follows:

```
    <target name="build" depends="prepare"
            description="build the model and application modules.">

        <ant dir="./model" target="package">
            <property name="outdir" value="${outdir}" />
            <property name="setProps" value="true" />
        </ant>

        <ant dir="./application" target="package">
            <property name="outdir" value="${outdir}" />
            <property name="setProps" value="true" />
        </ant>

    </target>
```

Thus, we can build both the application and model projects by running the main project. This may not seem like a big deal, but imagine a project with hundreds of subprojects that build thousands of components. Without a buildfile, such a project could become unmanageable. In fact, a project with just 10 to 20 components can benefit greatly from using nested buildfiles. We will use this same technique as we create the Web application in Chapter 6, the applet, and the EJB of this project. The master buildfile orchestrates the correct running order for all the subprojects. We could revisit this main project after we finish each additional subproject and update it. In the next section, we will discuss the applet buildfile.

The Applet Project

The applet project is a simple applet that reads the output of the HelloWorldServlet (defined in Chapter 6) and shows it in a JLabel. The dependency on the Web application is at runtime; there are no compile-time dependencies to the Web application. We'll discuss the applet here and the Web application in the next chapter.

Overview of the Applet Class

The meat of the applet implementation is in the init() method, as follows:

```
    public void init(){
```

```
    URL uGreeting;
    String sGreeting="Bye Bye";

    getAppletContext()
  .showStatus("Getting hello message from server.");

    try{
        uGreeting = new URL(
               getDocumentBase(),
               "HelloWorldServlet");

        sGreeting = getGreeting(uGreeting);
    }
    catch(Exception e){
        getAppletContext()
         .showStatus("Unable to communicate with server.");
        e.printStackTrace();
    }
    text.setText(sGreeting);

}
```

The init() method gets the document base (the URL from which the applet's page was loaded) URL from the applet context. Then, the init() method uses the document base and the URI identifying the HelloWorldServlet to create a URL that has the output of the HelloWorldServlet:

```
uGreeting = new URL( getDocumentBase(), "HelloWorldServlet");
```

It uses a helper method called getGreeting() to parse the output of the HelloWorldServlet, and then displays the greeting in the Applet's JLabel (text.setText(sGreeting);). The helper method is as follows:

```
private String getGreeting(URL uGreeting)throws Exception{
     String line;
     int endTagIndex;
     BufferedReader reader=null;
    . . .

 reader = new BufferedReader(
               new InputStreamReader (
                       uGreeting.openStream()));

        while((line=reader.readLine())!=null){
            System.out.println(line);

            if (line.startsWith("<h1>")){
                getAppletContext().showStatus("Parsing message.");
                endTagIndex=line.indexOf("</h1>");
```

```
                    line=line.substring(4,endTagIndex);
                    break;
                }
            }
          ...
        return line;
    }
```

Basically, the method gets the output stream from the URL (uGreeting.openStream()) and goes through the stream line by line looking for a line that begins with <h1>. Then, it pulls the text out of the <h1> tag.

The output of the HelloServlet looks like this:

```
<html>
<head>
<title>Hello World</title>
</head>
<body>
<h1>Hello World!</h1>
</body>
```

The code that the helper method retrieves appears in bold. Listing 5.4 shows the complete code for HelloWorldApplet.

```
package xptoolkit.applet;
import javax.swing.JApplet;
import javax.swing.JLabel;
import java.awt.Font;
import java.awt.BorderLayout;
import java.applet.AppletContext;
import java.net.URL;
import java.io.InputStreamReader;
import java.io.BufferedReader;

public class HelloWorldApplet extends javax.swing.JApplet {

    JLabel text;


    public HelloWorldApplet() {
        this.getContentPane().setLayout(new BorderLayout());
        text = new JLabel("Bye Bye");
        text.setAlignmentX(JLabel.CENTER_ALIGNMENT);
        text.setAlignmentY(JLabel.CENTER_ALIGNMENT);
        Font f = new Font("Arial", Font.BOLD, 20);
        text.setFont(f);
```

Listing 5.4 HelloWorldApplet that communicates with HelloWorldServlet. (continues)

```
        getContentPane().add(text,BorderLayout.CENTER);
    }


    public void init(){
        URL uGreeting;
        String sGreeting="Bye Bye";

        this.doLayout();
        getAppletContext()
.showStatus("Getting hello message from server.");

        try{
            uGreeting = new URL(
                   this.getDocumentBase(),
                   "HelloWorldServlet");

            sGreeting = getGreeting(uGreeting);
        }
        catch(Exception e){
            getAppletContext()
.showStatus("Unable to communicate with server.");
            e.printStackTrace();
        }
        text.setText(sGreeting);

    }

    private String getGreeting(URL uGreeting)throws Exception{
         String line;
         int endTagIndex;
         BufferedReader reader=null;


         try{
            reader = new BufferedReader(
                    new InputStreamReader (
                            uGreeting.openStream()));
            while((line=reader.readLine())!=null){
                System.out.println(line);
                if (line.startsWith("<h1>")){
                    getAppletContext().showStatus("Parsing message.");
                    endTagIndex=line.indexOf("</h1>");
                    line=line.substring(4,endTagIndex);
                    break;
                }
            }
```

Listing 5.4 HelloWorldApplet that communicates with HelloWorldServlet. (continues)

```
        }
        finally{
           if (reader!=null)reader.close();
        }
        return line;
    }

}
```

Listing 5.4 HelloWorldApplet that communicates with HelloWorldServlet.

Creating a Buildfile for the Applet

The applet project buildfile is quite simple (see Listing 5.5); it is structured much like the application project buildfile.

```
<project name="applet" default="all" >


    <target name="setProps" unless="setProps"
                          description="setup the properties.">
        <property name="outdir" value="/tmp/app" />
    </target>


    <target name="init" depends="setProps"
                          description="initialize the properties.">
        <tstamp/>

        <property name="local_outdir" value="${outdir}/applet" />
        <property name="build" value="${local_outdir}/classes" />
        <property name="lib" value="${outdir}/lib" />
        <property name="jar" value="${lib}/helloapplet.jar" />
    </target>

    <target name="clean" depends="init"
                          description="clean up the output directories.">
          <delete dir="${build}" />
          <delete dir="${jar}" />
    </target>


    <target name="prepare" depends="init"
                          description="prepare the output directory.">
```

Listing 5.5 Applet project's buildfile. (continues)

```
            <mkdir dir="${build}" />
            <mkdir dir="${lib}" />
      </target>

      <target name="compile" depends="prepare"
                            description="compile the Java source.">
          <javac srcdir="./src" destdir="${build}" />
      </target>

      <target name="package" depends="compile"
               description="package the Java classes into a jar.">
          <jar jarfile="${jar} "
             basedir="${build}" />
      </target>

      <target name="all" depends="clean,package"
                            description="perform all targets."/>

  </project>
```

Listing 5.5 Applet project's buildfile.

Building the Applet with Ant

To build the applet, we need to navigate to the Applet directory, set up the environment, and then run Ant at the command line. To clean the output from the build, we run "ant clean" at the command line. Both building and cleaning the applet are demonstrated as follows.

First we build the applet:

```
C:\CVS\...\MVCHelloWorld\Applet>ant
Buildfile: build.xml

setProps:

init:

clean:
   [delete] Deleting directory C:\tmp\app\lib

prepare:
    [mkdir] Created dir: C:\tmp\app\applet\classes
    [mkdir] Created dir: C:\tmp\app\lib

compile:
    [javac] Compiling 1 source file to C:\tmp\app\applet\classes
```

06 ch05.qxd 10/10/2001 9:38 AM Page 113

```
package:
      [jar] Building jar: C:\tmp\app\lib\helloapplet.jar

all:

BUILD SUCCESSFUL

Total time: 4 seconds
```

Now we clean the applet:

```
C:\CVS\...\MVCHelloWorld\Applet>ant clean
Buildfile: build.xml

setProps:

init:

clean:
   [delete] Deleting directory C:\tmp\app\applet\classes
   [delete] Deleting directory C:\tmp\app\lib

BUILD SUCCESSFUL

Total time: 0 seconds
```

Hello World Recap

It's important to recap what we have done. We created a common Java library called model.jar. This model.jar file is used by a Web application and a standalone executable Java application in an executable JAR file. We created an applet that can communicate with the Web application we create in Chapter 6. Once the applet loads into the browser, the applet communicates over HTTP to the Web application's HelloWorldServlet, which is covered in depth in Chapter 6.

In Chapter 6 we set up the GreetingFactory so that it talks to an enterprise session bean that in turn talks to an enterprise entity bean. The Web application buildfile is set up so that with the addition of a property file, it can talk to either the enterprise beans or to the local implementation in the model library.

Summary

In this chapter, we took a very complex project with a few components and subsystems, albeit a simple implementation, and built it in an orchestrated fashion. We showed how to create a Java application in an executable JAR and a Java applet in a JAR. We also demonstrated how to package a set of classes that is shared by more than one application in a JAR.