Server Setup and Configuration

Topics in This Chapter

- Installing and configuring Java
- Downloading and setting up a server
- Configuring your development environment
- Testing your setup
- Simplifying servlet and JSP deployment
- Locating files in Tomcat, JRun, and Resin
- Organizing projects into Web applications

Chapter 2

Before you can start learning specific servlet and JSP techniques, you need to have the right software and know how to use it. This introductory chapter explains how to obtain, configure, test, and use free versions of all the software needed to run servlets and JavaServer Pages (JSP). The initial setup involves seven steps, as outlined below.

1. **Download and install the Java Software Development Kit (SDK).** This step involves downloading an implementation of the Java 2 Platform, Standard Edition and setting your `PATH` appropriately. It is covered in Section 2.1.
2. **Download a server.** This step involves obtaining a server that implements the Servlet 2.3 (JSP 1.2) or Servlet 2.4 (JSP 2.0) APIs. It is covered in Section 2.2.
3. **Configure the server.** This step involves telling the server where the SDK is installed, changing the port to 80, and possibly making several server-specific customizations. The general approach is outlined in Section 2.3, with Sections 2.4–2.6 providing specific details for Apache Tomcat, Macromedia JRun, and Caucho Resin.
4. **Set up your development environment.** This step involves setting your `CLASSPATH` to include your top-level development directory and the JAR file containing the servlet and JSP classes. It is covered in Section 2.7.
5. **Test your setup.** This step involves checking the server home page and trying some simple JSP pages and servlets. It is covered in Section 2.8.

Please see updated setup information at
http://www.coreservlets.com/Apache-Tomcat-Tutorial/

6. **Establish a simplified deployment method.** This step involves choosing an approach for copying resources from your development directory to the server's deployment area. It is covered in Section 2.9.
7. **Create custom Web applications.** This step involves creating a separate directory for your application and modifying `web.xml` to give custom URLs to your servlets. This step can be postponed until you are comfortable with basic servlet and JSP development. It is covered in Section 2.11.

2.1 Download and Install the Java Software Development Kit (SDK)

You probably have already installed the Java Platform, but if not, doing so should be your first step. Current versions of the servlet and JSP APIs require the Java 2 Platform (Standard Edition—J2SE—or Enterprise Edition—J2EE). If you aren't using J2EE features like Enterprise JavaBeans (EJB) or Java Messaging Service (JMS), we recommend that you use the standard edition. Your server will supply the classes needed to add servlet and JSP support to Java 2 Standard Edition.

But what Java version do you need? Well, it depends on what servlet/JSP API you are using, and whether you are using a full J2EE-compliant application server (e.g., WebSphere, WebLogic, or JBoss) or a standalone servlet/JSP container (e.g., Tomcat, JRun, or Resin). If you are starting from scratch, we recommend that you use the latest Java version (1.4); doing so will give you the best performance and guarantee that you are compatible with future releases. But, if you want to know the minimum supported version, here is a quick summary.

- **Servlets 2.3 and JSP 1.2** (standalone servers). Java 1.2 or later.
- **J2EE 1.3** (which includes servlets 2.3 and JSP 1.2). Java 1.3 or later.
- **Servlets 2.4 and JSP 2.0** (standalone servers). Java 1.3 or later.
- **J2EE 1.4** (which includes servlets 2.4 and JSP 2.0). Java 1.4 or later.

We use Java 1.4 in our examples.

For Solaris, Windows, and Linux, obtain Java 1.4 at `http://java.sun.com/j2se/1.4/` and 1.3 at `http://java.sun.com/j2se/1.3/`. Be sure to download the SDK (Software Development Kit), not just the JRE (Java Runtime Environment)—the JRE is intended only for executing already compiled Java class files and lacks a compiler. For other platforms, check first whether a Java 2 implementation comes preinstalled as it does with MacOS X. If not, see Sun's list of third-party Java implementations at `http://java.sun.com/cgi-bin/java-ports.cgi`.

Your Java implementation should come with complete configuration instructions, but the key point is to set the PATH (not CLASSPATH!) environment variable to refer to the directory that contains java and javac, typically *java_install_dir*/bin. For example, if you are running Windows and installed the SDK in C:\j2sdk1.4.1_01, you might put the following line in your C:\autoexec.bat file. Remember that the autoexec.bat file is executed only when the system is booted.

```
set PATH=C:\j2sdk1.4.1_01\bin;%PATH%
```

If you want to download an already configured autoexec.bat file that contains the PATH setting and the other settings discussed in this chapter, go to http://www.coreservlets.com/, go to the source code archive, and select Chapter 2.

On Windows NT/2000/XP, you could also right-click on My Computer, select Properties, then Advanced, then Environment Variables. Then, you would update the PATH value and press the OK button.

On Unix (Solaris, Linux, etc.), if the SDK is installed in /usr/j2sdk1.4.1_01 and you use the C shell, you would put the following into your .cshrc file.

```
setenv PATH /usr/j2sdk1.4.1_01/bin:$PATH
```

After rebooting (Windows; not necessary if you set the variables interactively) or logging out and back in (Unix), verify that the Java setup is correct by opening a DOS window (Windows) or shell (Unix) and typing `java -version` and `javac -help`. You should see a real result *both* times, not an error message about an unknown command. Alternatively, if you use an Integrated Development Environment (IDE) like Borland JBuilder, Eclipse, IntelliJ IDEA, or Sun ONE Studio, compile and run a simple program to confirm that the IDE knows where you installed Java.

2.2 Download a Server for Your Desktop

Your second step is to download a server (often called a "servlet container" or "servlet engine") that implements the Servlet 2.3 Specification (JSP 1.2) or the Servlet 2.4 Specification (JSP 2.0) for use on your desktop. In fact, we typically keep *three* servers (Apache Tomcat, Macromedia JRun, and Caucho Resin) installed on our desktops and test applications on all the servers, to keep us aware of cross-platform deployment issues and to prevent us from accidentally using nonportable features. We'll give details on each of these servers throughout the book.

Regardless of the server that you use for final deployment, you will want at least one server *on your desktop* for development. Even if the deployment server is in the office next to you connected by a lightning-fast network connection, you still don't

want to use it for your development. Even a test server on your intranet that is inaccessible to customers is much less convenient for development purposes than a server right on your desktop. Running a development server on your desktop simplifies development in a number of ways, as compared to deploying to a remote server each and every time you want to test something. Here is why:

- **It is faster to test.** With a server on your desktop, there is no need to use FTP or another upload program. The harder it is for you to test changes, the less frequently you will test. Infrequent testing will let errors persist that will slow you down in the long run.
- **It is easier to debug.** When running on your desktop, many servers display the standard output in a normal window. This is in contrast to deployment servers on which the standard output is almost always either hidden or only available in a log file after execution is completed. So, with a desktop server, plain old `System.out.println` statements become useful tracing and debugging utilities.
- **It is simple to restart.** During development, you will find that you frequently need to restart the server or reload your Web application. For example, the server typically reads the `web.xml` file (see Section 2.11, "Web Applications: A Preview") only when the server starts or a server-specific command is given to reload a Web application. So, you normally have to restart the server or reload the Web application each time you modify `web.xml`. Even when servers have an interactive method of reloading `web.xml`, tasks such as clearing session data, resetting the `ServletContext`, or replacing modified class files used indirectly by servlets or JSP pages (e.g., beans or utility classes) may still necessitate that the server be restarted. Some older servers also need to be restarted because they implement servlet reloading unreliably. (Normally, servers instantiate the class that corresponds to a servlet only once and keep the instance in memory between requests. With *servlet reloading*, a server automatically replaces servlets that are in memory but whose class files have changed on the disk.) Besides, some deployment servers recommend completely disabling servlet reloading to increase performance. So, it is much more productive to develop in an environment in which you can restart the server or reload the Web application with a click of the mouse—without asking for permission from other developers who might be using the server.
- **It is more reliable to benchmark.** Although it is difficult to collect accurate timing results for short-running programs even in the best of circumstances, running benchmarks on multiuser systems that have heavy and varying system loads is notoriously unreliable.

- **It is under your control.** As a developer, you may not be the administrator of the system on which the test or deployment server runs. You might have to ask some system administrator every time you want the server restarted. Or, the remote system may be down for a system upgrade at the most critical juncture of your development cycle. Not fun.
- **It is easy to install.** Downloading and configuring a server takes no more than an hour. By using a server on your desktop instead of a remote one, you'll probably save yourself that much time the very first day you start developing.

If you can run the same server on your desktop that you use for deployment, all the better. So, if you are deploying on BEA WebLogic, IBM WebSphere, Oracle9i AS, etc., and your license permits you to also run the server on your desktop, by all means do so. But one of the beauties of servlets and JSP is that you don't *have* to; you can develop with one server and deploy with another.

Following are some of the most popular free options for desktop development servers. In all cases, the free version runs as a standalone Web server. In most cases, you have to pay for the deployment version that can be integrated with a regular Web server like Microsoft IIS, iPlanet/Sun ONE Server, Zeus, or the Apache Web Server. However, the performance difference between using one of the servers as a servlet and JSP engine within a regular Web server and using it as a complete standalone Web server is not significant enough to matter during development. See http://java.sun.com/products/servlet/industry.html for a more complete list of servers and server plugins that support servlets and JSP.

- **Apache Tomcat.** Tomcat 5 is the official reference implementation of the servlet 2.4 and JSP 2.0 specifications. Tomcat 4 is the official reference implementation for servlets 2.3 (JSP 1.2). Both versions can be used as standalone servers during development or can be plugged into a standard Web server for use during deployment. Like all Apache products, Tomcat is entirely free and has complete source code available. Of all the servers, it also tends to be the one that is most compliant with the latest servlet and JSP specifications. However, the commercial servers tend to be better documented, easier to configure, and faster. To download Tomcat, start at http://jakarta.apache.org/tomcat/, go to the binaries download section, and choose the latest release build of Tomcat.
- **Macromedia JRun.** JRun is a servlet and JSP engine that can be used in standalone mode for development or plugged into most common commercial Web servers for deployment. It is free for development purposes, but you must purchase a license before deploying with it. It is a popular choice among developers looking for easier administration than Tomcat. For details, see http://www.macromedia.com/software/jrun/.

- **Caucho's Resin.** Resin is a fast servlet and JSP engine with extensive XML support. Along with Tomcat and JRun, it is one of the three most popular servers used by commercial Web hosting companies that provide servlet and JSP support. It is free for development and noncommercial deployment purposes. For details, see http://caucho.com/products/resin/.
- **New Atlanta's ServletExec.** ServletExec is another popular servlet and JSP engine that can be used in standalone mode for development or, for deployment, plugged into the Microsoft IIS, Apache, and Sun ONE servers. You can download and use it for free, but some of the high-performance capabilities and administration utilities are disabled until you purchase a license. The ServletExec Debugger is the configuration you would use as a standalone desktop development server. For details, see http://www.newatlanta.com/products/servletexec/.
- **Jetty.** Jetty is an open-source server that supports servlets and JSP technology and is free for both development and deployment. It is often used as a complete standalone server (rather than integrated inside a non-Java Web server), even for deployment. For details, see http://jetty.mortbay.org/jetty/.

2.3 Configure the Server

Once you have downloaded and installed both the Java Platform itself and a server that supports servlets and JSP, you need to configure your server to run on your system. This configuration involves the following generic steps; the following three sections give specific details for Tomcat, JRun, and Resin.

Please note that these directions are geared toward using the server as a standalone Web server for use in desktop development. For deployment, you often set up your server to act as plugin within a traditional Web server like Apache or IIS. This configuration is beyond the scope of this book; use the wizard that comes with the server or read the configuration instructions in the vendor's documentation.

1. **Identifying the SDK installation directory.** To compile JSP pages, the server needs to know the location of the Java classes that are used by the Java compiler (e.g., `javac` or `jikes`). With most servers, either the server installation wizard detects the location of the SDK directory or you need to set the `JAVA_HOME` environment variable to refer to that directory. `JAVA_HOME` should list the base SDK installation directory, not the `bin` subdirectory.

2. **Specifying the port.** Most servers come preconfigured to use a non-standard port, just in case an existing server is already using port 80. If no server is already using port 80, for convenience, set your newly installed server to use that port.
3. **Making server-specific customizations.** These settings vary from server to server. Be sure to read your server's installation directions.

2.4 Configuring Apache Tomcat

Of all of the popular servlet and JSP engines, Tomcat is the hardest to configure. Tomcat is also the most fluid of the popular servers: compared to most other servers, Tomcat has more frequent releases and each version has more significant changes to the setup and configuration instructions. So, to handle new versions of Tomcat, we maintain an up-to-date Web page at http://www.coreservlets.com/ for installing and configuring Tomcat. Our online Tomcat configuration page includes sample versions of the three major files you need to edit: autoexec.bat, server.xml, and web.xml. If you use a version of Tomcat later than 4.1.24, you may want to refer to that Web site for details. Instructions consistent with release 4.1.24 follow.

Your first step is to download the Tomcat zip file from http://jakarta.apache.org/tomcat/. Click on Binaries and choose the latest release version. Assuming you are using JDK 1.4, select the "LE" version (e.g., tomcat-4.1.24-LE-jdk14.zip). Next, unzip the file into a location of your choosing. The only restriction is that the location cannot be protected from write access: Tomcat creates temporary files when it runs, so Tomcat must be installed in a location to which the user who starts Tomcat has write access. Unzipping Tomcat will result in a top-level directory similar to C:\jakarta-tomcat-4.1.24-LE-jdk14 (hereafter referred to as *install_dir*). Once you have downloaded and unzipped the Tomcat files, configuring the server involves the following steps. We give a quick summary below, then provide details in the following subsections.

1. **Setting the `JAVA_HOME` variable.** Set this variable to list the base SDK installation directory.
2. **Specifying the server port.** Edit *install_dir*/conf/server.xml and change the value of the `port` attribute of the `Connector` element from 8080 to 80.
3. **Enabling servlet reloading.** Add a `DefaultContext` element to *install_dir*/conf/server.xml to tell Tomcat to reload servlets that have been loaded into the server's memory but whose class files have changed on disk since they were loaded.

4. **Enabling the ROOT context.** To enable the default Web application, uncomment the following line in *install_dir*/conf/server.xml.
   `<Context path="" docBase="ROOT" debug="0"/>`
5. **Turning on the invoker servlet.** To permit you to run servlets without making changes to your web.xml file, some versions of Tomcat require you to uncomment the `/servlet/*` `servlet-mapping` element in *install_dir*/conf/web.xml.
6. **Increasing DOS memory limits.** On older Windows versions, tell the operating system to reserve more space for environment variables.
7. **Setting `CATALINA_HOME`.** Optionally, set the `CATALINA_HOME` environment variable to refer to the base Tomcat installation directory.

The following subsections give details on each of these steps. Please note that this section describes the use of Tomcat as a standalone server for servlet and JSP *development*. It requires a totally different configuration to deploy Tomcat as a servlet and JSP container integrated within a regular Web server (e.g., with `mod_webapp` in the Apache Web Server). For information on the use of Tomcat for deployment, see http://jakarta.apache.org/tomcat/tomcat-4.1-doc/.

Setting the JAVA_HOME Variable

The most critical Tomcat setting is the `JAVA_HOME` environment variable—an improper setting stops Tomcat from finding the classes used by `javac` and thus prevents Tomcat from handling JSP pages. This variable should list the base SDK installation directory, not the bin subdirectory. For example, if you are running Windows and you installed the SDK in C:\j2sdk1.4.1_01, you might put the following line in your C:\autoexec.bat file. Remember that the autoexec.bat file is executed only when the system is booted.

```
set JAVA_HOME=C:\j2sdk1.4.1_01
```

On Windows NT/2000/XP, you could also right-click on My Computer, select Properties, then Advanced, then Environment Variables. Then, you would enter the `JAVA_HOME` value and click OK.

On Unix (Solaris, Linux, MacOS X, AIX, etc.), if the SDK is installed in /usr/local/java1.4 and you use the C shell, you would put the following into your .cshrc file.

```
setenv JAVA_HOME /usr/local/java1.4
```

Rather than setting the `JAVA_HOME` environment variable globally in the operating system, some developers prefer to edit the Tomcat startup script and set the variable there. If you prefer this strategy, edit *install_dir*/bin/catalina.bat (Windows) and insert the following line at the top of the file, after the first set of comments.

```
set JAVA_HOME=C:\j2sdk1.4.1_01
```

Be sure to make a backup copy of catalina.bat before making the changes. Unix users would make similar changes to catalina.sh.

Specifying the Server Port

Most of the free servers listed in Section 2.2 use a nonstandard default port to avoid conflicts with other Web servers that may already be using the standard port (80). Tomcat is no exception: it uses port 8080 by default. However, if you are using Tomcat in standalone mode (i.e., as a complete Web server, not just as a servlet and JSP engine integrated within another Web server) and have no other server running permanently on port 80, you will find it more convenient to use port 80. That way, you don't have to use the port number in every URL you type in your browser. Note, however, that on Unix, you must have system administrator privileges to start services on port 80 or other port numbers below 1024. You probably have such privileges on your desktop machine; you do not necessarily have them on deployment servers. Furthermore, many Windows XP Professional implementations have Microsoft IIS already registered on port 80; you'll have to disable IIS if you want to run Tomcat on port 80. You can permanently disable IIS from the Administrative Tools/Internet Information Services section of the Control Panel.

Modifying the port number involves editing *install_dir*/conf/server.xml, changing the `port` attribute of the `Connector` element from 8080 to 80, and restarting the server. Replace *install_dir* with the base Tomcat installation location. For example, if you downloaded the Java 1.4 version of Tomcat 4.1.24 and unzipped it into the C directory, you would edit C:\jakarta-tomcat-4.1.24-LE-jdk14\conf\server.xml.

With Tomcat, the original element will look something like the following:

```
<Connector className="org.apache.coyote.tomcat4.CoyoteConnector"
           port="8080" minProcessors="5" maxProcessors="75"
           ... />
```

It should change to something like the following:

```
<Connector className="org.apache.coyote.tomcat4.CoyoteConnector"
           port="80" minProcessors="5" maxProcessors="75"
           ... />
```

Note that this element varies a bit from one Tomcat version to another. The easiest way to find the correct entry is to search for 8080 in server.xml; there should be only one noncomment occurrence. Be sure to make a backup of server.xml before you edit it, just in case you make a mistake that prevents the server from running. Also, remember that XML is case sensitive, so, for instance, you cannot replace `port` with `Port` or `Connector` with `connector`.

Enabling Servlet Reloading

The next step is to tell Tomcat to check the modification dates of the class files of requested servlets and reload ones that have changed since they were loaded into the server's memory. This slightly degrades performance in deployment situations, so is turned off by default. However, if you fail to turn it on for your development server, you'll have to restart the server or reload your Web application every time you recompile a servlet that has already been loaded into the server's memory.

To turn on servlet reloading, edit *install_dir*/conf/server.xml by adding a `DefaultContext` subelement to the main `Service` element and supply `true` for the `reloadable` attribute. The easiest way to do this is to find the following comment:

```
<!-- Define properties for each web application. ...
... -->
```

and insert the following line just below it:

```
<DefaultContext reloadable="true"/>
```

Again, be sure to make a backup copy of server.xml before making this change.

Enabling the ROOT Context

The ROOT context is the default Web application in Tomcat; it is convenient to use when you are first learning about servlets and JSP (although you'll use your own Web applications once you're more experienced—see Section 2.11). The default Web application is already enabled in Tomcat 4.0 and some versions of Tomcat 4.1. But, in Tomcat 4.1.24, it is disabled by default. To enable it, uncomment the following line in *install_dir*/conf/server.xml:

```
<Context path="" docBase="ROOT" debug="0"/>
```

Turning on the Invoker Servlet

The invoker servlet lets you run servlets without first making changes to the WEB-INF/web.xml file in your Web application. Instead, you just drop your servlet into WEB-INF/classes and use the URL http://*host*/servlet/*ServletName* (for the default Web application) or http://*host*/*webAppPrefix*/servlet/*ServletName* (for custom Web applications). The invoker servlet is extremely convenient when you are first learning and even when you are in the initial development phase of real projects. But, as discussed at length later in the book, you do not want it on at deployment time. Up until Apache Tomcat 4.1.12, the invoker was enabled by default. However, a security flaw was recently uncovered whereby the invoker servlet could be used to see the source

code of servlets that were generated from JSP pages. Although this may not matter in most cases, it might reveal proprietary code to outsiders, so, as of Tomcat 4.1.12, the invoker was disabled by default. We suspect that the Jakarta project will fix the problem soon and reenable the invoker servlet in upcoming Tomcat releases. In the meantime, however, you almost certainly want to enable it when learning. Just be sure that you do so only on a desktop development machine that is not accessible to the outside world.

To enable the invoker servlet, uncomment the following `servlet-mapping` element in *install_dir*/conf/web.xml. Note that the filename is web.xml, not server.xml, and do not confuse this Tomcat-specific web.xml file with the standard one that goes in the WEB-INF directory of each Web application.

```
<servlet-mapping>
  <servlet-name>invoker</servlet-name>
  <url-pattern>/servlet/*</url-pattern>
</servlet-mapping>
```

Increasing DOS Memory Limits

If you use an old version of Windows (i.e., Windows 98/Me or earlier), you may have to change the DOS memory settings for the startup and shutdown scripts. If you get an "Out of Environment Space" error message when you start the server, you will need to right-click on *install_dir*/bin/startup.bat, select Properties, select Memory, and change the Initial Environment entry from Auto to at least 2816. Repeat the process for *install_dir*/bin/shutdown.bat.

Setting CATALINA_HOME

In some cases, it is also helpful to set the `CATALINA_HOME` environment variable to refer to the base Tomcat installation directory. This variable identifies the location of various Tomcat files to the server. However, if you are careful to avoid copying the server startup and shutdown scripts and instead use only shortcuts (called "symbolic links" on Unix) instead, you are *not* required to set this variable. See Section 2.9 (Establish a Simplified Deployment Method) for more information on using these shortcuts.

Testing the Basic Server Setup

To verify that you have configured Tomcat successfully, double-click on *install_dir*/bin/startup.bat (Windows) or execute *install_dir*/bin/startup.sh (Unix/Linux). Open a browser and enter http://localhost/ (http://localhost:8080/

if you chose not to change the port to 80). You should see something similar to Figure 2–1. Shut down the server by double-clicking on *install_dir*/bin/shutdown.bat (Windows) or executing *install_dir*/bin/shutdown.sh (Unix). If you cannot get Tomcat to run, try going to *install_dir*/bin and typing `catalina run`; this will prevent Tomcat from starting a separate window and will let you see error messages such as those that stem from the port being in use or `JAVA_HOME` being defined incorrectly.

After you customize your development environment (see Section 2.7), be sure to perform the more exhaustive tests listed in Section 2.8.

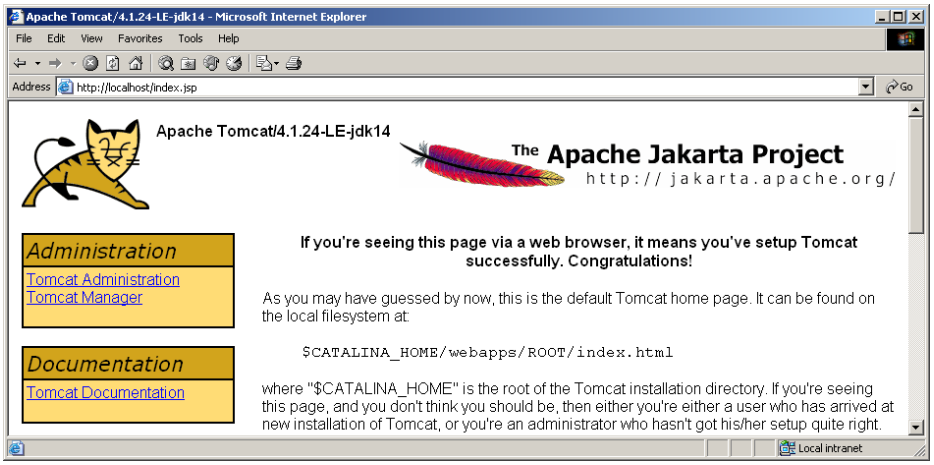

Figure 2–1 Tomcat home page.

2.5 Configuring Macromedia JRun

To use JRun on your desktop, your first step is to download the free development version of JRun from http://www.macromedia.com/software/jrun/ and run the installation wizard. Most of the configuration settings are specified during installation. There are seven main settings you are likely to want to specify. The following list gives a quick summary; details are given in the subsections that follow the list.

1. **The serial number.** Leave it blank for the free development server.
2. **User restrictions.** You can limit the use of JRun to your account or make it available to anyone on your system.
3. **The SDK installation location.** Specify the base directory, not the bin subdirectory.

4. **The server installation directory.** In most cases, you just accept the default.
5. **The administrator username and password.** You will need these values for making additional customizations later.
6. **The autostart capability.** During development, you do *not* want JRun to start automatically. In particular, on Windows, you should *not* identify JRun as a Windows service.
7. **The server port.** You will probably want to change it from 8100 to 80.

The JRun Serial Number

Using JRun in development mode (i.e., where only requests from the local machine are accepted) does not require a serial number. So, unless you are using a full deployment version of the server, leave the serial number blank when prompted for it. You can upgrade to a deployment version later without reinstalling the server. See Figure 2–2.

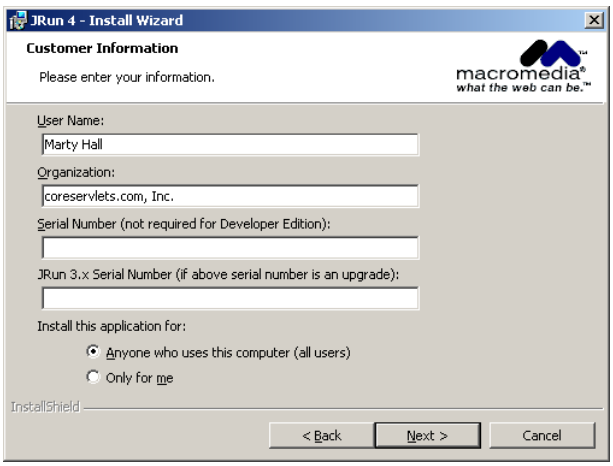

Figure 2–2 Omit the serial number if you are using the free development version of JRun.

JRun User Restrictions

When you install JRun, you will be asked whether you want the server to be available to all users on your system or only to your account. See Figure 2–2. Select whichever is appropriate.

The Java Installation Location

The installation wizard will search for a Java installation and present its base directory as the default choice. If that choice refers to your most recent Java version, accept the default. However, if the installation wizard finds an older version of Java, choose Browse and select an alternative location. In such a case, make sure you supply the location of the base directory, not the bin subdirectory. Also, be sure that you designate the location of the full SDK (called "JDK" in Java 1.3 and earlier), not of the JRE (Java Runtime Environment)—the JRE directory lacks the classes needed to compile JSP pages. See Figure 2–3.

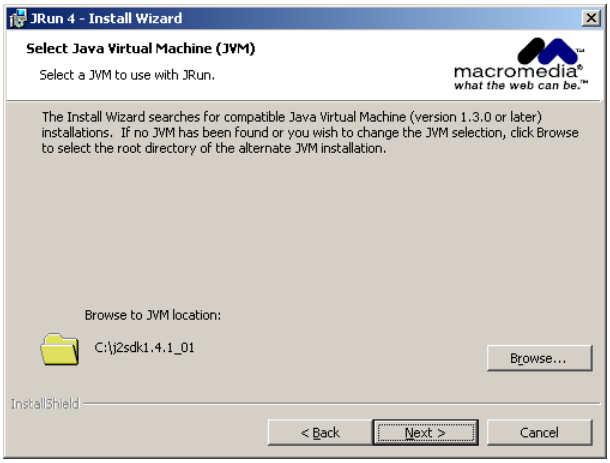

Figure 2–3 Be sure the JVM location refers to the base installation directory of your latest Java version.

The Server Installation Location

You can choose whatever directory you want for this option. Most users simply accept the default, which, on Windows, is C:\JRun4.

The Administrator Username and Password

The installation wizard will prompt you for a name and password. The values you supply are arbitrary, but be sure to remember what you specified; you will need them to customize the server later. See Figure 2–4.

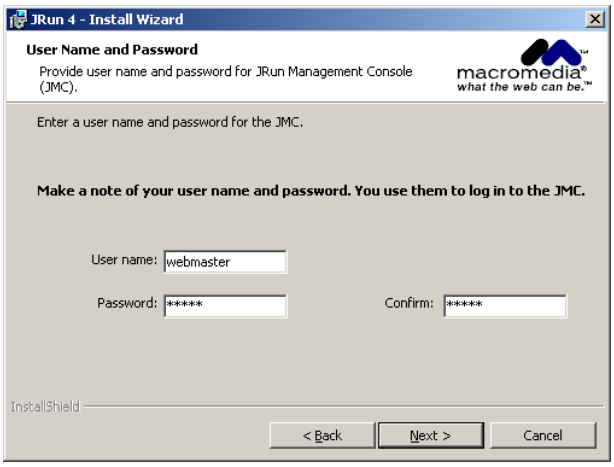

Figure 2–4 Be sure to remember the administrator username and password.

The Autostart Capability

When using JRun as a development server, you will find it much more convenient to start and stop JRun manually than to have the operating system start JRun automatically. So, when prompted whether you want JRun to be a Windows service, leave the choice unchecked. See Figure 2–5.

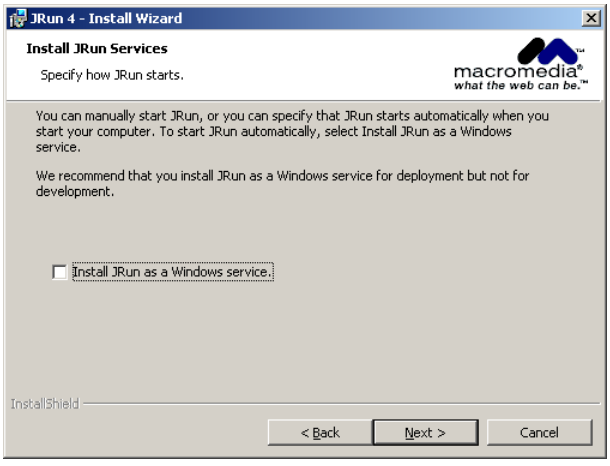

Figure 2–5 Do *not* install JRun as a Windows service.

The Server Port

After completing the installation, go to the Start menu, select Programs, select Macromedia JRun 4, and choose JRun Launcher. Select the admin server and press Start. Do the same for the default server. See Figure 2–6.

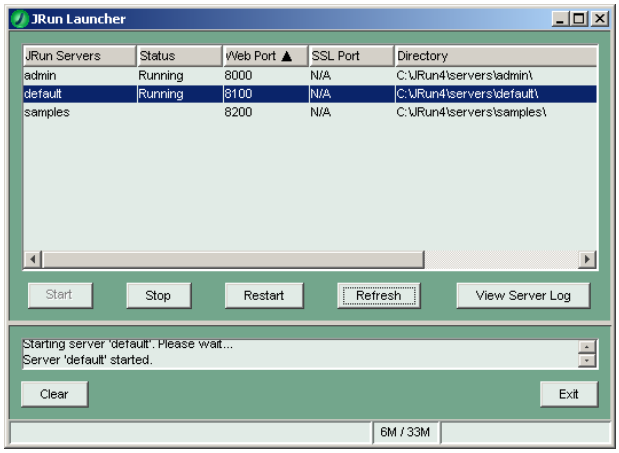

Figure 2–6 You use the JRun Launcher to start and stop the administration and default servers.

Next, either open a browser and enter the URL http://localhost:8000/ or go to the Start menu, select Programs, select Macromedia JRun 4, and choose JRun Management Console. Either option will result in a Web page that prompts you for a username and password. Enter the values you specified during installation, then select Services under the default server in the left-hand pane. This will yield a result similar to Figure 2–7. Next, choose WebService, change the port from 8100 to 80, press Apply, and stop and restart the server.

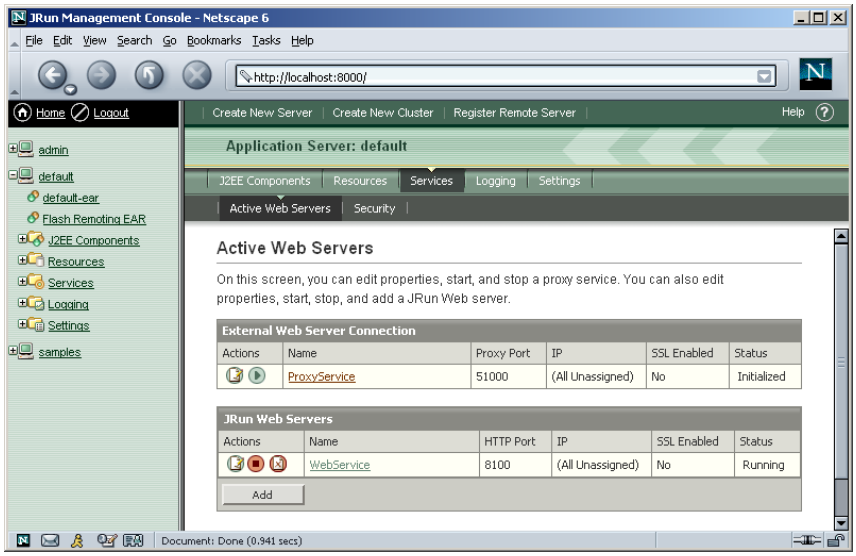

Figure 2–7 After selecting Services under the default server, choose WebService to edit the port of the default JRun server.

Testing the Basic Server Setup

To verify that you have configured JRun successfully, open the JRun Launcher by going to the Start menu, selecting Programs, choosing Macromedia JRun 4, and designating JRun Launcher. If you just changed the server port, the server is probably already running. (Note that you do not need to start the admin server unless you want to modify additional server options.) Open a browser and enter `http://localhost/` (`http://localhost:8100/` if you chose not to change the port to 80). You should see something similar to Figure 2–8. Shut down the server by pressing Stop in the JRun Launcher.

After you customize your development environment (see Section 2.7), be sure to perform the more exhaustive tests listed in Section 2.8.

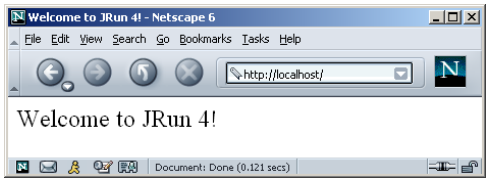

Figure 2–8 The JRun home page.

2.6 Configuring Caucho Resin

To run Resin on your desktop, you should first download the Resin zip file from http://caucho.com/products/resin/ and unzip it into a location of your choosing (hereafter referred to as *install_dir*). Once you have done so, configuring the server involves two simple steps.

1. **Setting the `JAVA_HOME` variable.** Set this variable to list the base SDK installation directory.
2. **Specifying the port.** Edit *install_dir*/conf/resin.conf and change the value of the `port` attribute of the `http` element from 8080 to 80.

Details are given in the following subsections.

Setting the JAVA_HOME Variable

The most important setting is the `JAVA_HOME` environment variable. This variable should refer to the base SDK installation directory. Details are given in Section 2.4 (Configuring Apache Tomcat), but for a quick example, if you are using Java 1.4.1 on Windows, you might put the following line in C:\autoexec.bat.

```
set JAVA_HOME=C:\j2sdk1.4.1_01
```

Specifying the Resin Port

To avoid conflicts with preexisting servers, Resin uses port 8080 by default. However, if you won't be simultaneously running another server, you will probably find it convenient to change Resin to use port 80, the standard HTTP port. To do this, edit *install_dir*/conf/resin.conf and change `<http port='8080'/>` to `<http port='80'/>`.

Testing the Basic Server Setup

To verify that you have configured Resin successfully, double-click on *install_dir*/bin/httpd.exe. Open a browser and enter http://localhost/ (http://localhost:8080/ if you chose not to change the port to 80). You should see something similar to Figure 2–9. Shut down the server by selecting Stop in the small dialog box that pops up when you start the server.

After you customize your development environment (see Section 2.7), be sure to perform the more exhaustive tests listed in Section 2.8.

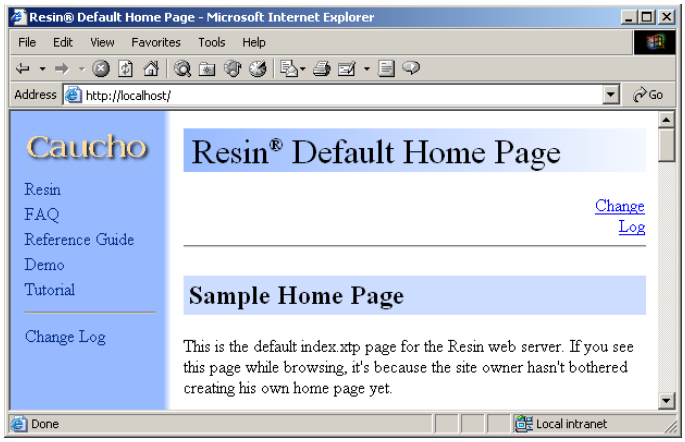

Figure 2–9 Resin home page.

2.7 Set Up Your Development Environment

You configured and tested the server, so you're all set, right? Well, no, not quite. That's just the local *deployment* environment. You still have to set up your personal *development* environment. Otherwise, you won't be able to compile servlets and auxiliary Java classes that you write. Configuring your development environment involves the following steps.

1. **Creating a development directory.** Choose a location in which to develop your servlets, JSP documents, and supporting classes.
2. **Setting your `CLASSPATH`.** Tell the compiler about the servlet and JSP JAR file and the location of your development directory. *Setting this variable incorrectly is the single most common cause of problems for beginners.*
3. **Making shortcuts to start and stop the server.** Make sure it is convenient to start and stop the server.
4. **Bookmarking or installing the servlet and JSP API documentation.** You'll refer to this documentation frequently, so keep it handy.

The following subsections give details on each of these steps.

Creating a Development Directory

The first thing you should do is create a directory in which to place the servlets and JSP documents that you develop. This directory can be in your home directory (e.g., ~/ServletDevel on Unix) or in a convenient general location (e.g., C:\ServletDevel on Windows). It should *not*, however, be in the server's installation directory.

Eventually, you will organize this development directory into different Web applications (each with a common structure—see Section 2.11, "Web Applications: A Preview"). For initial testing of your environment, however, you can just put servlets either directly in the development directory (for packageless servlets) or in a subdirectory that matches the servlet package name. After compiling, you can simply copy the class files to the server's default Web application.

Many developers put all their code in the server's deployment directory (see Section 2.10). We strongly discourage this practice and instead recommend one of the approaches described in Section 2.9 (Establish a Simplified Deployment Method). Although developing in the deployment directory seems simpler at the beginning since it requires no copying of files, it significantly complicates matters in the long run. Mixing development and deployment locations makes it hard to separate an operational version from a version you are testing, makes it difficult to test on multiple servers, and makes organization much more complicated. Besides, your desktop is almost certainly not the final deployment server, so you'll eventually have to develop a good system for deploying anyhow.

Core Warning

Don't use the server's deployment directory as your development location. Instead, keep a separate development directory.

Setting Your CLASSPATH

Since servlets and JSP are not part of the Java 2 Platform, Standard Edition, you must identify the servlet classes to the compiler. The *server* already knows about the servlet classes, but the *compiler* (i.e., `javac`) you use for development probably doesn't. So, if you don't set your `CLASSPATH`, attempts to compile servlets, tag libraries, or other classes that use the servlet API will fail with error messages about unknown classes. The exact location of the servlet JAR file varies from server to server. In most cases, you can hunt around in the *install_dir*/lib directory. Or, read your server's documentation to discover the location. Once you find the JAR file, add the location to your development `CLASSPATH`. Here are the locations for some common development servers:

- **Tomcat.**
  install_dir/common/lib/servlet.jar
- **JRun.**
  install_dir/lib/jrun.jar
- **Resin.**
  install_dir/lib/jsdk23.jar

In addition to the servlet JAR file, you also need to put your development directory in the CLASSPATH. Although this is not necessary for simple packageless servlets, once you gain experience you will almost certainly use packages. Compiling a file that is in a package and that uses another class in the same package requires the CLASSPATH to include the directory that is at the top of the package hierarchy. In this case, that's the development directory we discussed in the first subsection. Forgetting this setting is perhaps *the* most common mistake made by beginning servlet programmers.

Core Approach

Remember to add your development directory to your CLASSPATH. Otherwise, you will get "Unresolved symbol" error messages when you attempt to compile servlets that are in packages and that make use of other classes in the same package.

Finally, you should include "." (the current directory) in the CLASSPATH. Otherwise, you will only be able to compile packageless classes that are in the top-level development directory.

Here are a few representative methods of setting the CLASSPATH. They assume that your development directory is C:\ServletDevel (Windows) or /usr/ServletDevel (Unix) and that you are using Tomcat 4. Replace *install_dir* with the actual base installation location of the server. Be sure to use the appropriate case for the filenames; they are case sensitive (even on a Windows platform!). If a Windows path contains spaces (e.g., C:\Documents and Settings*Your Name*\My Documents\...), enclose it in double quotes. Note that these examples represent only one approach for setting the CLASSPATH. For example, you could create a script that invokes `javac` with a designated value for the `-classpath` option. In addition, many Java integrated development environments have a global or project-specific setting that accomplishes the same result. But those settings are totally IDE specific and aren't discussed here.

- **Windows 95/98/Me.** Put the following in C:\autoexec.bat. (Note that this all goes on one line with no spaces—it is broken here for readability.)
```
set CLASSPATH=.;
              C:\ServletDevel;
              install_dir\common\lib\servlet.jar
```
- **Windows NT/2000/XP.** Use the autoexec.bat file as above, or right-click on My Computer, select Properties, then System, then Advanced, then Environment Variables. Then, enter the CLASSPATH value from the previous bullet and click OK.
- **Unix (C shell).** Put the following in your .cshrc. (Again, in the real file it goes on a single line without spaces.)
```
setenv CLASSPATH .:
                 /usr/ServletDevel:
                 install_dir/common/lib/servlet.jar
```

Making Shortcuts to Start and Stop the Server

During our development, we find ourselves frequently restarting the server. As a result, we find it convenient to place shortcuts to the server startup and shutdown icons inside the main development directory or on the desktop. You will likely find it convenient to do the same.

For example, for Tomcat on Windows, go to *install_dir*/bin, right-click on startup.bat, and select Copy. Then go to your development directory, right-click in the window, and select Paste Shortcut (not just Paste). Repeat the process for *install_dir*/bin/shutdown.bat. Some users like to put the shortcuts on the desktop or their Start menu. If you put the shortcuts there, you can even right-click on the shortcut, select Properties, then enter a keyboard shortcut by typing a key in the "Keyboard shortcut" text field. That way, you can start and stop the server just by pressing Control-Alt-*SomeKey* on your keyboard.

On Unix, you would use `ln -s` to make a symbolic link to startup.sh, tomcat.sh (needed even though you don't directly invoke this file), and shutdown.sh.

For JRun on Windows, go to the Start menu, select Programs, select Macromedia JRun 4, right-click on the JRun Launcher icon, and select Copy. Then go to your development directory, right-click in the window, and select Paste Shortcut (not just Paste). Repeat the process for the JRun Management Console if you so desire. There is no separate shutdown icon; the JRun Launcher lets you both start and stop the server.

For Resin on Windows, right-click on *install_dir*/bin/httpd.exe, and select Copy. Then go to your development directory, right-click in the window, and select Paste Shortcut (not just Paste). There is no separate shutdown icon; invoking httpd.exe results in a popup window with a Quit button that lets you stop the server.

Bookmarking or Installing the Servlet and JSP API Documentation

Just as no serious programmer should develop general-purpose Java applications without access to the Java 1.4 or 1.3 API documentation (in Javadoc format), no serious programmer should develop servlets or JSP pages without access to the API for classes in the `javax.servlet` packages. Here is a summary of where to find the API. (Remember that the source code archive at http://www.coreservlets.com/ has up-to-date links to all URLs cited in the book, in addition to the source code for all examples.)

- **http://java.sun.com/products/jsp/download.html**
  This site lets you download the Javadoc files for the servlet 2.4 (JSP 2.0) or servlet 2.3 (JSP 1.2) APIs. You will probably find this API so useful that it will be worth having a local copy instead of browsing it online. However, some servers bundle this documentation, so check before downloading. (See the next bullet.)
- **On your local server**
  Some servers come bundled with the servlet and JSP Javadocs. For example, with Tomcat, you can access the API by going to the default home page (http://localhost/) and clicking on Tomcat Documentation and then Servlet/JSP Javadocs. Or, bookmark *install_dir*/webapps/tomcat-docs/catalina/docs/api/index.html; doing so lets you access the documentation even when Tomcat is not running. Neither JRun nor Resin bundles the API, however.
- **http://java.sun.com/products/servlet/2.3/javadoc/**
  This site lets you browse the servlet 2.3 API online.
- **http://java.sun.com/j2ee/sdk_1.3/techdocs/api/**
  This address lets you browse the complete API for version 1.3 of the Java 2 Platform, Enterprise Edition (J2EE), which includes the servlet 2.3 and JSP 1.2 packages.
- **http://java.sun.com/j2ee/1.4/docs/api/**
  This address lets you browse the complete API for version 1.4 of the Java 2 Platform, Enterprise Edition (J2EE), which includes the servlet 2.4 and JSP 2.0 packages.

2.8 Test Your Setup

Before trying your own servlets or JSP pages, you should make sure that the SDK, the server, and your development environment are all configured properly. Verification involves the three steps summarized below; more details are given in the subsections following the list.

1. **Verifying your SDK installation.** Be sure that both `java` and `javac` work properly.
2. **Checking your basic server configuration.** Access the server home page, a simple user-defined HTML page, and a simple user-defined JSP page.
3. **Compiling and deploying some simple servlets.** Try a basic packageless servlet, a servlet that uses packages, and a servlet that uses both packages and a utility (helper) class.

Verifying Your SDK Installation

Open a DOS window (Windows) or shell (Unix) and type `java -version` and `javac -help`. You should see a real result *both* times, not an error message about an unknown command. Alternatively, if you use an Integrated Development Environment (IDE), compile and run a simple program to confirm that the IDE knows where you installed Java. If either of these tests fails, review Section 2.1 (Download and Install the Java Software Development Kit (SDK)) and double-check the installation instructions that came with the SDK.

Checking Your Basic Server Configuration

First, start the server and access the standard home page (http://localhost/, or http://localhost:*port*/ if you did not change the port to 80). If this fails, review the instructions of Sections 2.3–2.6 and double-check your server's installation instructions.

After you have verified that the server is running, you should make sure that you can install and access simple HTML and JSP pages. This test, if successful, shows two important things. First, successfully accessing an HTML page shows that you understand which directories should hold HTML and JSP files. Second, successfully accessing a new JSP page shows that the Java compiler (not just the Java virtual machine) is configured properly.

Eventually, you will almost certainly want to create and use your own Web applications (see Section 2.11, "Web Applications: A Preview"), but for initial testing we recommend that you use the default Web application. Although Web applications follow a common directory structure, the exact location of the default Web application is server specific. Check your server's documentation for definitive instructions, but we summarize the locations for Tomcat, JRun, and Resin in the following list. Where we list *SomeDirectory* you can use any directory name you like. (But you are never allowed to use WEB-INF or META-INF as directory names. For the default Web application, you also must avoid a directory name that matches the URL prefix of any existing Web application such as samples or examples.) If you are running on your local machine, you can use localhost where we list *host* in the URLs.

- **Tomcat HTML/JSP directory.**
  install_dir/webapps/ROOT
  (or *install_dir*/webapps/ROOT/*SomeDirectory*)
- **JRun HTML/JSP directory.**
  install_dir/servers/default/default-ear/default-war
  (or *install_dir*/servers/default/default-ear/default-war/*SomeDirectory*)
- **Resin HTML/JSP directory.**
  install_dir/doc
  (or *install_dir*/doc/*SomeDirectory*)
- **Corresponding URLs.**
  http://*host*/Hello.html
  (or http://*host*/*SomeDirectory*/Hello.html)
  http://*host*/Hello.jsp
  (or http://*host*/*SomeDirectory*/Hello.jsp)

For your first tests, we suggest you simply drop Hello.html (Listing 2.1, Figure 2–10) and Hello.jsp (Listing 2.2, Figure 2–11) into the appropriate locations. For now, don't worry about what the JSP document does; we'll cover that later. The code for these files, as well as *all* the code from the book, is available online at http://www.coreservlets.com/. That Web site also contains links to all URLs cited in the book, updates, additions, information on training courses, and other servlet and JSP resources. It also contains a frequently updated page on Tomcat configuration (since Tomcat changes more often than the other servers).

If neither the HTML file nor the JSP file works (e.g., you get File Not Found—404—errors), you probably are either using the wrong directory for the files or misspelling the URL (e.g., using a lowercase h in Hello.jsp). If the HTML file works but the JSP file fails, you probably have incorrectly specified the base SDK directory (e.g., with the JAVA_HOME variable) and should review Section 2.7 (Set Up Your Development Environment).

Listing 2.1 Hello.html

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">
<HTML>
<HEAD><TITLE>HTML Test</TITLE></HEAD>
<BODY BGCOLOR="#FDF5E6">
<H1>HTML Test</H1>
Hello.
</BODY></HTML>
```

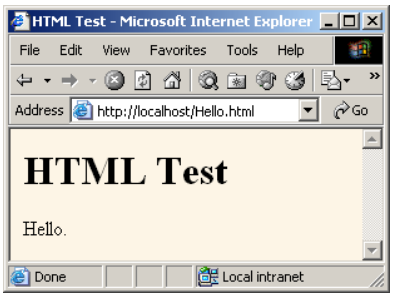

Figure 2–10 Result of Hello.html.

Listing 2.2 Hello.jsp

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">
<HTML>
<HEAD><TITLE>JSP Test</TITLE></HEAD>
<BODY BGCOLOR="#FDF5E6">
<H1>JSP Test</H1>
Time: <%= new java.util.Date() %>
</BODY>
</HTML>
```

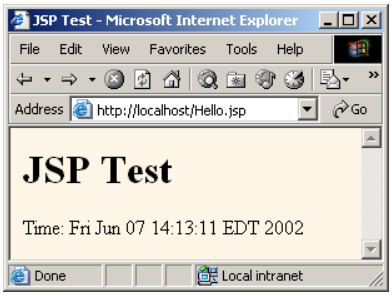

Figure 2–11 Result of Hello.jsp.

Compiling and Deploying Some Simple Servlets

OK, so your development environment is all set. At least you *think* it is. It would be nice to confirm that hypothesis. Following are three test servlets that help verify it.

Test 1: A Servlet That Does Not Use Packages

The first servlet to try is a basic one: no packages, no utility (helper) classes, just simple HTML output. Rather than writing your own test servlet, you can just download HelloServlet.java (Listing 2.3) from the book's source code archive at http://www.coreservlets.com/. Again, don't worry about how this servlet works—that is covered in detail in the next chapter—the point here is just to test your setup. If you get compilation errors, go back and check your CLASSPATH settings (Section 2.7)—you most likely erred in listing the location of the JAR file that contains the servlet classes (e.g., servlet.jar).

Once you compile HelloServlet.java, put HelloServlet.class in the appropriate location (usually the WEB-INF/classes directory of your server's default Web application). Check your server's documentation for this location, or see the following list for a summary of the locations used by Tomcat, JRun, and Resin. Then, access the servlet with the URL http://*host*/servlet/HelloServlet (or http://*host*:*port*/servlet/HelloServlet if you chose not to change the port number as described in Section 2.3). Use localhost for *host* if you are running the server on your desktop system. You should get something similar to Figure 2–12. If this URL fails but the test of the server itself succeeded, you probably put the class file in the wrong directory.

Notice that you use servlet (not servlets!) in the URL even though there is no real directory named servlet. URLs of the form .../servlet/*ServletName* are just an instruction to a special servlet (called the *invoker servlet*) to run the servlet with the specified name. The servlet code itself is in any of the locations the server normally uses (usually, .../WEB-INF/classes for individual class files or .../WEB-INF/lib for JAR files that contain servlets). Using default URLs like this is convenient during your initial development, but once you are ready to deploy, you will almost certainly disable this capability and register a separate URL for each servlet. See Section 2.11 (Web Applications: A Preview) for details. In fact, servers are not strictly required to support these default URLs, and some of the high-end application servers, most notably BEA WebLogic, do not.

- **Tomcat directory for Java .class files.**
  install_dir/webapps/ROOT/WEB-INF/classes
  (Note: in many Tomcat versions, you'll have to manually create the classes directory.)
- **JRun directory for Java .class files.**
  install_dir/servers/default/default-ear/default-war/WEB-INF/classes
- **Resin directory for Java .class files.**
  install_dir/doc/WEB-INF/classes
- **Corresponding URL.**
  http://*host*/servlet/HelloServlet

Listing 2.3 HelloServlet.java

```
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;

/** Simple servlet used to test server. */

public class HelloServlet extends HttpServlet {
  public void doGet(HttpServletRequest request,
                    HttpServletResponse response)
      throws ServletException, IOException {
    response.setContentType("text/html");
    PrintWriter out = response.getWriter();
    String docType =
      "<!DOCTYPE HTML PUBLIC \"-//W3C//DTD HTML 4.0 " +
      "Transitional//EN\">\n";
    out.println(docType +
                "<HTML>\n" +
                "<HEAD><TITLE>Hello</TITLE></HEAD>\n" +
                "<BODY BGCOLOR=\"#FDF5E6\">\n" +
                "<H1>Hello</H1>\n" +
                "</BODY></HTML>");
  }
}
```

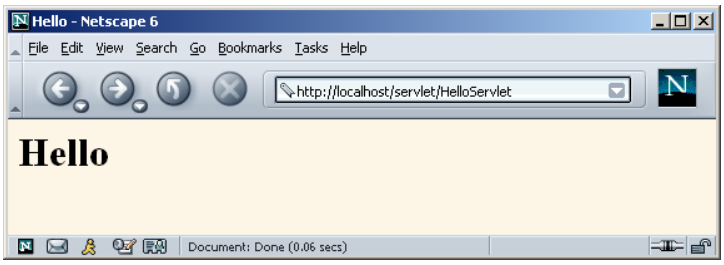

Figure 2–12 Result of http://localhost/servlet/HelloServlet.

Test 2: A Servlet That Uses Packages

The second servlet to try is one that uses packages but no utility classes. Packages are the standard mechanism for preventing class name conflicts in the Java programming language. There are three standard rules to remember:

1. **Insert package declarations in the code.** If a class is in a package, it must have "`package packageName;`" as the first noncomment line in the source code.
2. **Use a directory that matches the package name.** If a class is in a package, it must be in a directory that matches its package name. This is true for class files in both development and deployment locations.
3. **From Java code, use dots after packages.** When you refer to classes that are in packages either from within Java code or in a URL, you use a dot, not a slash, between the package name and the class name.

Again, rather than writing your own test, you can grab HelloServlet2.java (Listing 2.4) from the book's source code archive at http://www.coreservlets.com/. Since this servlet is in the `coreservlets` package, it should go in the coreservlets directory, *both* during development *and* when deployed to the server. If you get compilation errors, go back and check your `CLASSPATH` settings (Section 2.7)—you most likely forgot to include "." (the current directory). Once you compile HelloServlet2.java, put HelloServlet2.class in the coreservlets subdirectory of whatever directory the server uses for servlets that are not in custom Web applications (usually the WEB-INF/classes directory of the default Web application). Check your server's documentation for this location, or see the following list for a summary of the locations for Tomcat, JRun, and Resin. For now, you can simply copy the class file from the development directory to the deployment directory, but Section 2.9 (Establish a Simplified Deployment Method) provides some options for simplifying the process.

Once you have placed the servlet in the proper directory, access it with the URL http://localhost/servlet/coreservlets.HelloServlet2. Note that there is a dot, not a slash, between the package name and the servlet name in the URL. You should get something similar to Figure 2–13. If this test fails, you probably either typed the URL wrong (e.g., failed to maintain the proper case) or put HelloServlet2.class in the wrong location (e.g., directly in the server's WEB-INF/classes directory instead of in the coreservlets subdirectory).

- **Tomcat directory for packaged Java classes.**
  install_dir/webapps/ROOT/WEB-INF/classes/coreservlets
- **JRun directory for packaged Java classes.**
  install_dir/servers/default/default-ear/default-war/WEB-INF/classes/coreservlets
- **Resin directory for packaged Java classes.**
  install_dir/doc/WEB-INF/classes/coreservlets
- **Corresponding URL.**
  http://*host*/servlet/coreservlets.HelloServlet2

Listing 2.4 coreservlets/HelloServlet2.java

```
package coreservlets;

import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;

/** Simple servlet for testing the use of packages. */

public class HelloServlet2 extends HttpServlet {
  public void doGet(HttpServletRequest request,
                    HttpServletResponse response)
      throws ServletException, IOException {
    response.setContentType("text/html");
    PrintWriter out = response.getWriter();
    String docType =
      "<!DOCTYPE HTML PUBLIC \"-//W3C//DTD HTML 4.0 " +
      "Transitional//EN\">\n";
    out.println(docType +
                "<HTML>\n" +
                "<HEAD><TITLE>Hello (2)</TITLE></HEAD>\n" +
                "<BODY BGCOLOR=\"#FDF5E6\">\n" +
                "<H1>Hello (2)</H1>\n" +
                "</BODY></HTML>");
  }
}
```

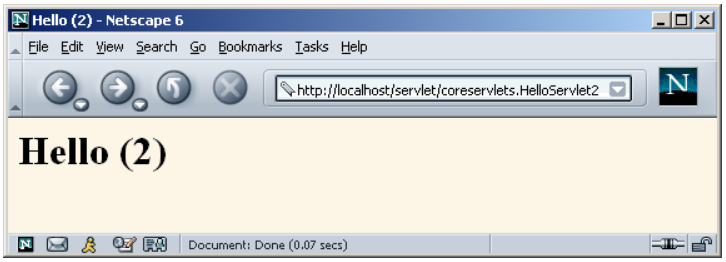

Figure 2–13 Result of http://localhost/servlet/coreservlets.HelloServlet2. Note that it is a dot, not a slash, between the package name and the class name.

Test 3: A Servlet That Uses Packages and Utilities

The final servlet you should test to verify the configuration of your server and development environment is one that uses both packages and utility classes. Listing 2.5 presents HelloServlet3.java, a servlet that uses the `ServletUtilities` class (Listing 2.6) to simplify the generation of the `DOCTYPE` (specifies the HTML version—useful when using HTML validators) and `HEAD` (specifies the title) portions of the HTML page. Those two parts of the page are useful (technically required, in fact) but are tedious to generate with servlet `println` statements. Again, the source code can be found at http://www.coreservlets.com/.

Since both the servlet and the utility class are in the `coreservlets` package, they should go in the coreservlets directory. If you get compilation errors, go back and check your `CLASSPATH` settings (Section 2.7)—you most likely forgot to include the top-level development directory. We've said it before, but we'll say it again: your `CLASSPATH` must include the top-level directory of your package hierarchy before you can compile a packaged class that makes use of another class that is in the same package or in any other user-defined (nonsystem) package. This requirement is not particular to servlets; it is the way packages work on the Java platform in general. Nevertheless, many servlet developers are unaware of this fact, and it is one of the (perhaps *the*) most common problems that beginning developers encounter. Furthermore, as we will see later, you *must* put all utility classes you write into packages if you want to use them from JSP pages, so virtually all the auxiliary classes (and most of the servlets) you write will be in packages. You might as well get used to the process of using packages now.

Core Warning

Your `CLASSPATH` must include your top-level development directory. Otherwise, you will get "unresolved symbol" errors when you attempt to compile servlets that are in packages and that also use user-defined classes that are in packages.

Once you compile HelloServlet3.java (which will automatically cause ServletUtilities.java to be compiled), put HelloServlet3.class and ServletUtilities.class in the coreservlets subdirectory of whatever directory the server uses for servlets that are not in custom Web applications (usually the WEB-INF/classes directory of the default Web application). Check your server's documentation for this location, or see the

following list for a summary of the locations used by Tomcat, JRun, and Resin. Then, access the servlet with the URL http://localhost/servlet/coreservlets.HelloServlet3. You should get something similar to Figure 2–14.

- **Tomcat directory for packaged Java classes.**
  install_dir/webapps/ROOT/WEB-INF/classes/coreservlets
- **JRun directory for packaged Java classes.**
  install_dir/servers/default/default-ear/default-war/WEB-INF/classes/coreservlets
- **Resin directory for packaged Java classes.**
  install_dir/doc/WEB-INF/classes/coreservlets
- **Corresponding URL.**
  http://*host*/servlet/coreservlets.HelloServlet3

Listing 2.5 coreservlets/HelloServlet3.java

```
package coreservlets;

import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;

/** Simple servlet for testing the use of packages
 *  and utilities from the same package.
 */

public class HelloServlet3 extends HttpServlet {
  public void doGet(HttpServletRequest request,
                    HttpServletResponse response)
      throws ServletException, IOException {
    response.setContentType("text/html");
    PrintWriter out = response.getWriter();
    String title = "Hello (3)";
    out.println(ServletUtilities.headWithTitle(title) +
                "<BODY BGCOLOR=\"#FDF5E6\">\n" +
                "<H1>" + title + "</H1>\n" +
                "</BODY></HTML>");
  }
}
```

Listing 2.6 coreservlets/ServletUtilities.java (Excerpt)

```
package coreservlets;

import javax.servlet.*;
import javax.servlet.http.*;

/** Some simple time savers. Note that most are static methods. */

public class ServletUtilities {
  public static final String DOCTYPE =
    "<!DOCTYPE HTML PUBLIC \"-//W3C//DTD HTML 4.0 " +
    "Transitional//EN\">";

  public static String headWithTitle(String title) {
    return(DOCTYPE + "\n" +
           "<HTML>\n" +
           "<HEAD><TITLE>" + title + "</TITLE></HEAD>\n");
  }

  ...
}
```

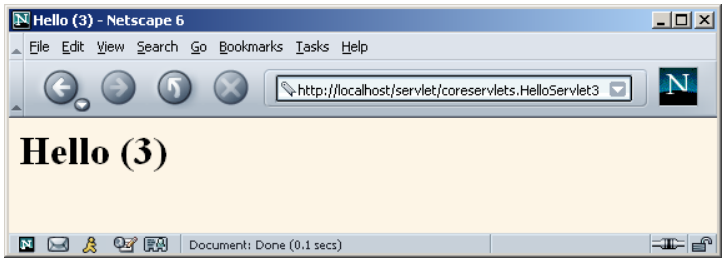

Figure 2–14 Result of http://localhost/servlet/coreservlets.HelloServlet3.

2.9 Establish a Simplified Deployment Method

OK, so you have a development directory. You can compile servlets with or without packages. You know which directory the servlet classes belong in. You know the URL that should be used to access them (at least the default URL; in Section 2.11, "Web Applications: A Preview," you'll see how to customize that address). But how do you

move the class files from the development directory to the deployment directory? Copying each one by hand every time is tedious and error prone. Once you start using Web applications (see Section 2.11), copying individual files becomes even more cumbersome.

There are several ways to simplify the process. Here are a few of the most popular ones. If you are just beginning with servlets and JSP, you probably want to start with the first option and use it until you become comfortable with the development process. Note that we do *not* list the option of putting your code directly in the server's deployment directory. Although this is one of the most common choices among beginners, it scales so poorly to advanced tasks that we recommend you steer clear of it from the start.

1. **Copying to a shortcut or symbolic link.**
2. **Using the `-d` option of `javac`.**
3. **Letting your IDE take care of deployment.**
4. **Using `ant` or a similar tool.**

Details on these four options are given in the following subsections.

Copying to a Shortcut or Symbolic Link

On Windows, go to the server's default Web application, right-click on the `classes` directory, and select Copy. Then go to your development directory, right-click, and select Paste Shortcut (not just Paste). Now, whenever you compile a packageless servlet, just drag the class files onto the shortcut. When you develop in packages, use the *right* mouse button to drag the entire directory (e.g., the `coreservlets` directory) onto the shortcut, release the mouse button, and select Copy. See Figure 2–15 for an example setup that simplifies testing of this chapter's examples on Tomcat, JRun, and Resin. On Unix, you can use symbolic links (created with `ln -s`) in a manner similar to that for Windows shortcuts.

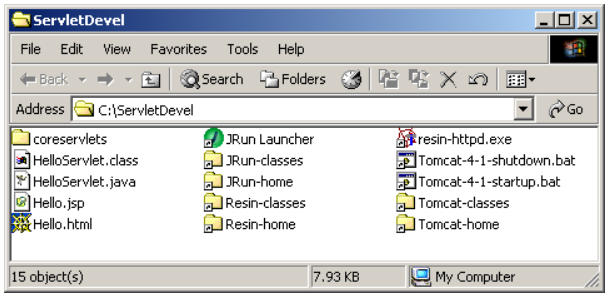

Figure 2–15 Using shortcuts to simplify deployment.

An advantage of this approach is that it is simple. So, it is good for beginners who want to concentrate on learning servlets and JSP, not deployment tools. Another advantage is that a variation applies once you start using your own Web applications (see Section 2.11). Just make a shortcut to the main Web application directory (typically one level up from the top of the default Web application), and copy the entire Web application each time by using the right mouse button to drag the directory that contains your Web application onto this shortcut and selecting Copy.

One disadvantage of this approach is that it requires repeated copying if you use multiple servers. For example, we keep three different servers (Tomcat, JRun, and Resin) on our development system and regularly test the code on all three servers. A second disadvantage is that this approach copies both the Java source code files and the class files to the server, whereas only the class files are needed. This may not matter much on your desktop server, but when you get to the "real" deployment server, you won't want to include the source code files.

Using the -d Option of javac

By default, the Java compiler (`javac`) places class files in the same directory as the source code files that they came from. However, `javac` has an option (`-d`) that lets you designate a different location for the class files. You need only specify the top-level directory for class files—`javac` will automatically put packaged classes in subdirectories that match the package names. So, for example, with Tomcat you could compile the `HelloServlet2` servlet (Listing 2.4, Section 2.8) as follows (line break added only for clarity; omit it in real life).

```
javac -d install_dir/webapps/ROOT/WEB-INF/classes
         HelloServlet2.java
```

You could even make a Windows batch file or Unix shell script or alias that makes a command like `servletc` expand to `javac -d install_dir/.../classes`. See http://java.sun.com/j2se/1.4/docs/tooldocs/win32/javac.html for more details on `-d` and other `javac` options.

An advantage of this approach is that it requires no manual copying of class files. Furthermore, the exact same command can be used for classes in different packages since `javac` automatically puts the class files in a subdirectory matching the package.

The main disadvantage is that this approach applies only to Java class files; it won't work for deploying HTML and JSP pages, much less entire Web applications.

Letting Your IDE Take Care of Deployment

Most servlet- and JSP-savvy development environments (e.g., IBM WebSphere Studio Application Developer, Sun ONE Studio, Borland JBuilder, Eclipse) have options that let you specify where to deploy class files for your project. Then, when

you tell the IDE to build the project, the class files are automatically deployed to the proper location (package-specific subdirectories and all).

An advantage of this approach, at least in some IDEs, is that it can deploy HTML and JSP pages and even entire Web applications, not just Java class files. A disadvantage is that it is an IDE-specific technique and thus is not portable across systems.

Using ant or a Similar Tool

Developed by the Apache foundation, `ant` is a tool similar to the Unix `make` utility. However, `ant` is written in the Java programming language (and thus is portable) and is touted to be both simpler to use and more powerful than `make`. Many servlet and JSP developers use `ant` for compiling and deploying. The use of `ant` is especially popular among Tomcat users and with those developing Web applications (see Section 2.11). Use of `ant` is discussed in Volume 2 of this book.

For general information on using `ant`, see http://jakarta.apache.org/ant/manual/. See http://jakarta.apache.org/tomcat/tomcat-4.1-doc/appdev/processes.html for specific guidance on using `ant` with Tomcat.

The main advantage of this approach is flexibility: `ant` is powerful enough to handle everything from compiling the Java source code to copying files to producing Web archive (WAR) files (see Section 2.11, "Web Applications: A Preview"). The disadvantage of `ant` is the overhead of learning to use it; there is a steeper learning curve with `ant` than with the other techniques in this section.

2.10 Deployment Directories for Default Web Application: Summary

The following subsections summarize the way to deploy and access HTML files, JSP pages, servlets, and utility classes in Apache Tomcat, Macromedia JRun, and Caucho Resin. The summary assumes that you are deploying files in the default Web application, have changed the port number to 80 (see Section 2.3), and are accessing servlets through the default URL (i.e., http://host/servlet/ServletName). Section 2.11 explains how to deploy user-defined Web applications and how to customize the URLs. But you'll probably want to start with the defaults just to confirm that everything is working properly. The Appendix (Server Organization and Structure) gives a unified summary of the directories used by Tomcat, JRun, and Resin for both the default Web application and custom Web applications.

If you are using a server on your desktop, you can use localhost for the *host* portion of each of the URLs in this section.

Tomcat

HTML and JSP Pages

- **Main Location.**
  install_dir/webapps/ROOT
- **Corresponding URLs.**
  http://*host*/*SomeFile*.html
  http://*host*/*SomeFile*.jsp
- **More Specific Location (Arbitrary Subdirectory).**
  install_dir/webapps/ROOT/*SomeDirectory*
- **Corresponding URLs.**
  http://*host*/*SomeDirectory*/*SomeFile*.html
  http://*host*/*SomeDirectory*/*SomeFile*.jsp

Individual Servlet and Utility Class Files

- **Main Location (Classes without Packages).**
  install_dir/webapps/ROOT/WEB-INF/classes
- **Corresponding URL (Servlets).**
  http://*host*/servlet/*ServletName*
- **More Specific Location (Classes in Packages).**
  install_dir/webapps/ROOT/WEB-INF/classes/*packageName*
- **Corresponding URL (Servlets in Packages).**
  http://*host*/servlet/*packageName.ServletName*

Servlet and Utility Class Files Bundled in JAR Files

- **Location.**
  install_dir/webapps/ROOT/WEB-INF/lib
- **Corresponding URLs (Servlets).**
  http://*host*/servlet/*ServletName*
  http://*host*/servlet/*packageName.ServletName*

JRun

HTML and JSP Pages

- **Main Location.**
  install_dir/servers/default/default-ear/default-war
- **Corresponding URLs.**
  http://*host*/*SomeFile*.html
  http://*host*/*SomeFile*.jsp

- **More Specific Location (Arbitrary Subdirectory).**
  install_dir/servers/default/default-ear/default-war/*SomeDirectory*
- **Corresponding URLs.**
  http://*host*/*SomeDirectory*/*SomeFile*.html
  http://*host*/*SomeDirectory*/*SomeFile*.jsp

Individual Servlet and Utility Class Files

- **Main Location (Classes without Packages).**
  install_dir/servers/default/default-ear/default-war/WEB-INF/classes
- **Corresponding URL (Servlets).**
  http://*host*/servlet/*ServletName*
- **More Specific Location (Classes in Packages).**
  install_dir/servers/default/default-ear/default-war/WEB-INF/classes/*packageName*
- **Corresponding URL (Servlets in Packages).**
  http://*host*/servlet/*packageName.ServletName*

Servlet and Utility Class Files Bundled in JAR Files

- **Location.**
  install_dir/servers/default/default-ear/default-war/WEB-INF/lib
- **Corresponding URLs (Servlets).**
  http://*host*/servlet/*ServletName*
  http://*host*/servlet/*packageName.ServletName*

Resin

HTML and JSP Pages

- **Main Location.**
  install_dir/doc
- **Corresponding URLs.**
  http://*host*/*SomeFile*.html
  http://*host*/*SomeFile*.jsp
- **More Specific Location (Arbitrary Subdirectory).**
  install_dir/doc/*SomeDirectory*
- **Corresponding URLs.**
  http://*host*/*SomeDirectory*/*SomeFile*.html
  http://*host*/*SomeDirectory*/*SomeFile*.jsp

Individual Servlet and Utility Class Files

- **Main Location (Classes without Packages).**
  install_dir/doc/WEB-INF/classes
- **Corresponding URL (Servlets).**
  http://*host*/servlet/*ServletName*
- **More Specific Location (Classes in Packages).**
  install_dir/doc/WEB-INF/classes/*packageName*
- **Corresponding URL (Servlets in Packages).**
  http://*host*/servlet/*packageName.ServletName*

Servlet and Utility Class Files Bundled in JAR Files

- **Location.**
  install_dir/doc/WEB-INF/lib
- **Corresponding URLs (Servlets).**
  http://*host*/servlet/*ServletName*
  http://*host*/servlet/*packageName.ServletName*

2.11 Web Applications: A Preview

Up to this point, we've been using the server's default Web application for our servlets. Most servers come preinstalled with a default Web application, and most servers let you invoke servlets in that application with URLs of the form http://*host*/servlet/*ServletName* or http://*host*/servlet/*packageName.ServletName*. Use of the default Web application and URL is very convenient when you are learning how to use servlets; you probably want to stick with these defaults when you first practice the techniques described throughout the book. So, if you are new to servlet and JSP development, skip this section for now.

However, once you have learned the basics of both servlets and JSP and are ready to start on real applications, you'll want to use your own Web application instead of the default one. Web applications are discussed in great detail in Volume 2 of this book, but a quick preview of the basics is presented in this section.

Core Approach

When first learning, use the default Web application and default servlet URLs. For serious applications, use custom Web applications and URLs that are assigned in the deployment descriptor (web.xml).

Most servers (including the three used as examples in this book) have server-specific administration consoles that let you create and register Web applications from within a Web browser. These consoles are discussed in Volume 2; for now, we restrict ourselves to the basic manual approach that is nearly identical on all servers. The following list summarizes the steps; the subsections that follow the steps give details.

1. **Make a directory whose structure mirrors the structure of the default Web application.** HTML (and, eventually, JSP) documents go in the top-level directory, the web.xml file goes in the WEB-INF subdirectory, and servlets and other classes go either in WEB-INF/classes or in a subdirectory of WEB-INF/classes that matches the package name.
2. **Update your `CLASSPATH`.** Add webAppDir/WEB-INF/classes to it.
3. **Register the Web application with the server.** Tell the server where the Web application directory (or JAR file created from it) is located and what prefix in the URL (see the next item) should be used to invoke the application. For example, with Tomcat, just drop the Web application directory in *install_dir*/webapps and then restart the server. The name of the directory becomes the Web application prefix.
4. **Use the designated URL prefix to invoke servlets or HTML/JSP pages from the Web application.** Invoke unpackaged servlets with a default URL of http://*host*/***webAppPrefix***/servlet/*ServletName*, packaged servlets with http://*host*/***webAppPrefix***/servlet/*packageName.ServletName*, and HTML pages from the top-level Web application directory with http://*host*/***webAppPrefix***/*filename*.html.
5. **Assign custom URLs for all your servlets.** Use the `servlet` and `servlet-mapping` elements of web.xml to give a URL of the form http://*host*/*webAppPrefix*/*someName* to each servlet.

Making a Web Application Directory

To make a Web application, create a directory in your development folder. That new directory should have the same general layout as the default Web application:

- HTML and, eventually, JSP documents go in the top-level directory (or any subdirectory other than WEB-INF).
- The web.xml file (sometimes called "the deployment descriptor") goes in the WEB-INF subdirectory.
- Servlets and other classes go either in WEB-INF/classes or, more commonly, in a subdirectory of WEB-INF/classes that matches the package name.

The easiest way to make such a directory is to copy an existing Web application. For instance, with Tomcat, you could copy the ROOT directory to your development

folder and rename it to testApp, resulting in something like C:\Servlets+JSP\testApp. As with the default Web application, we strongly advise against developing directly in the server's Web application directory. Keep a separate directory, and deploy it whenever you are ready to test. The easiest deployment option is to simply copy the directory to the server's standard location, but Section 2.9 (Establish a Simplified Deployment Method) gives several other alternatives.

Updating Your CLASSPATH

Recall from Section 2.7 (Set Up Your Development Environment) that your CLASSPATH needs to contain the top-level directory of .class files. This is true whether or not you are using custom Web applications, so add webAppDir/classes to the CLASSPATH.

Registering the Web Application with the Server

In this step, you tell the server where the Web application directory (or JAR file created from it) is located and what prefix in the URL (see the next subsection) should be used to invoke the application. There are various server-specific mechanisms for doing this registration, many of which involve the use of an interactive administration console. But, on most servers, you can also register a Web application simply by dropping the Web application directory in a standard location and then restarting the server. In such a case, the name of the Web application directory is used as the URL prefix. Here are the standard locations for Web application directories with the three servers used throughout the book.

- **Tomcat Web application autodeploy directory.**
  install_dir/webapps
- **JRun Web application autodeploy directory.**
  install_dir/servers/default
- **Resin Web application autodeploy directory.**
  install_dir/webapps

For example, we created a directory called testApp with the following structure:

- testApp/Hello.html
  The sample HTML file of Section 2.8 (Listing 2.1).
- testApp/Hello.jsp
  The sample JSP file of Section 2.8 (Listing 2.2).
- testApp/WEB-INF/classes/HelloServlet.class
  The sample packageless servlet of Section 2.8 (Listing 2.3).
- testApp/WEB-INF/classes/coreservlets/HelloServlet2.class
  The first sample packaged servlet of Section 2.8 (Listing 2.4).

- testApp/WEB-INF/classes/coreservlets/HelloServlet3.class
  The second sample packaged servlet of Section 2.8 (Listing 2.5).
- testApp/WEB-INF/classes/coreservlets/ServletUtilities.class
  The utility class (Listing 2.6) used by `HelloServlet3`.

WAR Files

Web ARchive (WAR) files provide a convenient way of bundling Web applications in a single file. Having a single large file instead of many small files makes it easier to transfer the Web application from server to server.

A WAR file is really just a JAR file with a .war extension, and you use the normal `jar` command to create it. For example, to bundle the entire testApp Web app into a WAR file named testApp2.war, you would just change directory to the testApp directory and execute the following command.

```
jar cvf testApp2.war *
```

There are a few options you can use in advanced applications (we discuss these in Volume 2 of the book), but for simple WAR files, that's it!

Again, the exact details of deployment are server dependent, but most servers let you simply drop a WAR file in the autodeploy directory, and the base name of the WAR file becomes the Web application prefix. For example, you would drop testApp2.war into the same directory you dropped testApp, restart the server, then invoke the test resources shown in Figures 2–16 through 2–20 by merely changing testApp to testApp2 in the URLs.

Using the URL Prefix

When you use Web applications, a special prefix is part of all URLs. For example:

- Unpackaged servlets are invoked with a default URL of
  http://*host*/***webAppPrefix***/servlet/*ServletName*
- Packaged servlets are invoked with
  http://*host*/***webAppPrefix***/servlet/*packageName.ServletName*
- Registered servlets (see the next subsection) are invoked with
  http://*host*/***webAppPrefix***/*customName*
- HTML pages from the top-level Web application directory are invoked with http://*host*/***webAppPrefix***/*filename*.html.
- HTML pages from subdirectories are invoked with
  http://*host*/***webAppPrefix***/*subdirectoryName*/*filename*.html.
- JSP pages are placed in the same locations as HTML pages and invoked in the same way (except that the file extension is .jsp instead of .html).

Most servers let you choose arbitrary prefixes, but, by default, the name of the directory (or the base name of the WAR file) becomes the Web application prefix. For example, we copied the testApp directory to the appropriate Web application directory (*install_dir*/webapps for Tomcat and Resin, *install_dir*/servers/default for JRun) and restarted the server. Then, we invoked the resources by using URLs identical to those of Section 2.8 except for the addition of testApp after the hostname. See Figures 2–16 through 2–20.

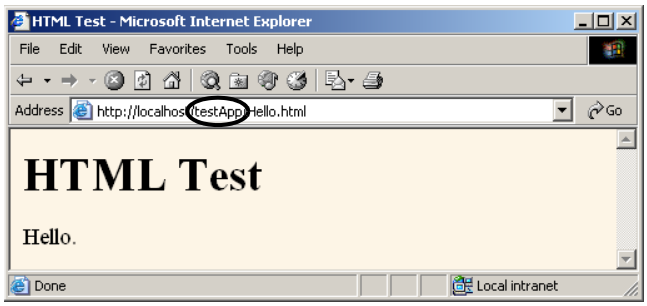

Figure 2–16 Hello.html invoked within a Web application.

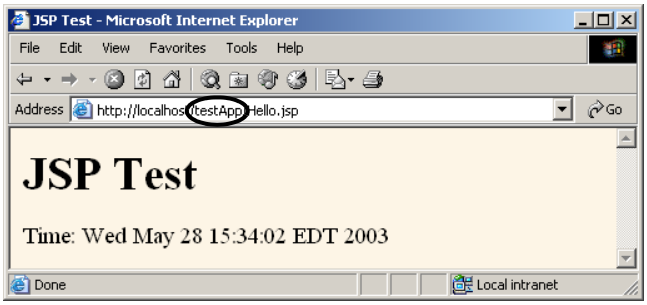

Figure 2–17 Hello.jsp invoked within a Web application.

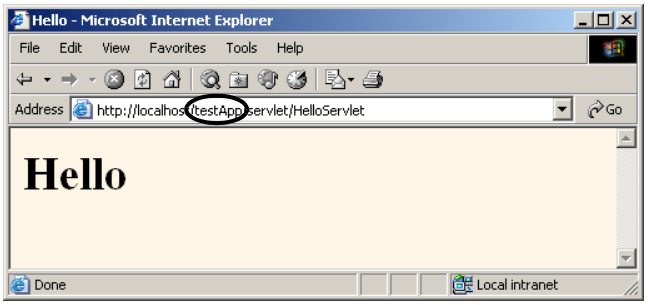

Figure 2–18 HelloServlet.class invoked with the default URL within a Web application.

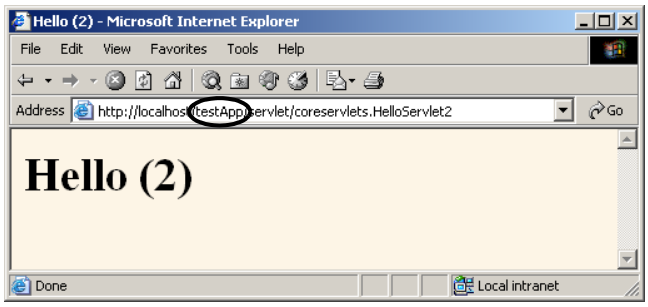

Figure 2–19 HelloServlet2.class invoked with the default URL within a Web application.

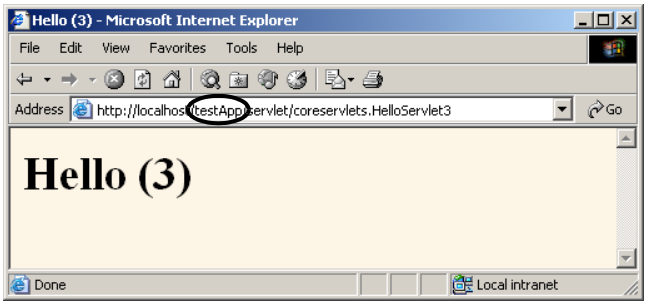

Figure 2–20 HelloServlet3.class invoked with the default URL within a Web application.

Assigning Custom URLs to Your Servlets

During initial development, it is very convenient to drop a servlet in WEB-INF/classes and immediately invoke it with http://*host*/*webAppPrefix*/servlet/*ServletName*. For deploying serious applications, however, you always want to define custom URLs.

You assign the URLs by using the `servlet` and `servlet-mapping` elements of web.xml (the deployment descriptor). The web.xml file is discussed in great detail in Volume 2 of the book, but for the purpose of registering custom URLs, you simply need to know five things:

- **The file location.** It *always* goes in WEB-INF.
- **The base form.** It starts with an XML header and a `DOCTYPE` declaration, and it contains a `web-app` element.
- **The way to give names to servlets.** You use `servlet` with `servlet-name` and `servlet-class` subelements.
- **The way to give URLs to named servlets.** You use `servlet-mapping` with `servlet-name` and `url-pattern` subelements.
- **When the web.xml file is read.** It is read *only* when the server starts.

Locating the Deployment Descriptor

The web.xml file always goes in the WEB-INF directory of your Web application. That is the *only* portable location; other locations (e.g., *install_dir*/conf for Tomcat) are nonstandard server extensions that you should steer clear of.

Defining the Base Format

A web.xml file that is compatible with both servlets 2.3 (JSP 1.2) and servlets 2.4 (JSP 2.0) has the following basic form:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!DOCTYPE web-app
  PUBLIC "-//Sun Microsystems, Inc.//DTD Web Application 2.3//EN"
  "http://java.sun.com/dtd/web-app_2_3.dtd">
<web-app>

</web-app>
```

Deployment descriptors that are specific to servlets 2.4 (JSP 2.0) are discussed in Volume 2 of the book.

Naming Servlets

To name a servlet, you use the `servlet` element within `web-app`, with `servlet-name` (any name) and `servlet-class` (the fully qualified class name) subelements. For example, to give the name `Servlet2` to `HelloServlet2`, you would use the following.

```
<servlet>
  <servlet-name>Servlet2</servlet-name>
  <servlet-class>coreservlets.HelloServlet2</servlet-class>
</servlet>
```

Giving URLs

To give a URL to a named servlet, you use the `servlet-mapping` element, with `servlet-name` (the previously assigned name) and `url-pattern` (the URL suffix, starting with a slash) subelements. For example, to give the URL http://*host*/*webAppPrefix*/servlet2 to the servlet named `Servlet2`, you would use the following.

```
<servlet-mapping>
  <servlet-name>Servlet2</servlet-name>
  <url-pattern>/servlet2</url-pattern>
</servlet-mapping>
```

Note that *all* the `servlet` elements must come before *any* of the `servlet-mapping` elements: you cannot intermingle them.

Reading the Deployment Descriptor

Many servers have "hot deploy" capabilities or methods to interactively restart Web applications. For example, JRun automatically restarts Web applications whose web.xml files have changed. By default, however, the web.xml file is read *only* when the server starts. So, unless you make use of a server-specific feature, you have to restart the server every time you modify the web.xml file.

Example

Listing 2.7 gives the full web.xml file for the testApp Web application. The file was placed in the WEB-INF directory of testApp, the testApp directory was copied to the server's Web application directory (e.g., *install_dir*/webapps for Tomcat and Resin, *install_dir*/servers/default for JRun), and the server was restarted. Figures 2–21 through 2–23 show the three sample servlets invoked with the registered URLs.

Listing 2.7 WEB-INF/web.xml

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!DOCTYPE web-app
   PUBLIC "-//Sun Microsystems, Inc.//DTD Web Application 2.3//EN"
   "http://java.sun.com/dtd/web-app_2_3.dtd">
<web-app>
  <servlet>
    <servlet-name>Servlet1</servlet-name>
    <servlet-class>HelloServlet</servlet-class>
  </servlet>
  <servlet>
    <servlet-name>Servlet2</servlet-name>
    <servlet-class>coreservlets.HelloServlet2</servlet-class>
  </servlet>
  <servlet>
    <servlet-name>Servlet3</servlet-name>
    <servlet-class>coreservlets.HelloServlet3</servlet-class>
  </servlet>
  <servlet-mapping>
    <servlet-name>Servlet1</servlet-name>
    <url-pattern>/servlet1</url-pattern>
  </servlet-mapping>
  <servlet-mapping>
    <servlet-name>Servlet2</servlet-name>
    <url-pattern>/servlet2</url-pattern>
  </servlet-mapping>
```

Listing 2.7 WEB-INF/web.xml *(continued)*

```
  <servlet-mapping>
    <servlet-name>Servlet3</servlet-name>
    <url-pattern>/servlet3</url-pattern>
  </servlet-mapping>
</web-app>
```

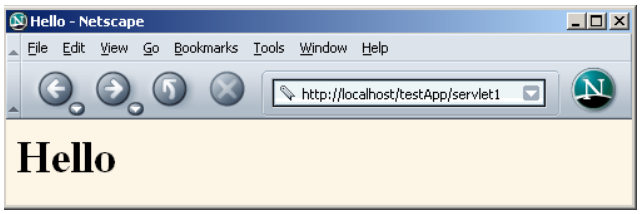

Figure 2–21 `HelloServlet` invoked with a custom URL.

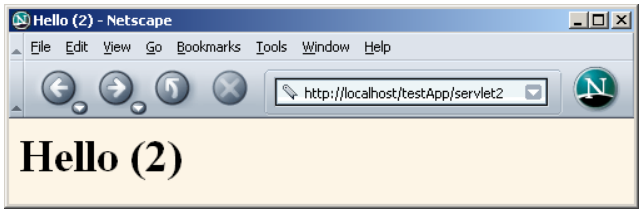

Figure 2–22 `HelloServlet2` invoked with a custom URL.

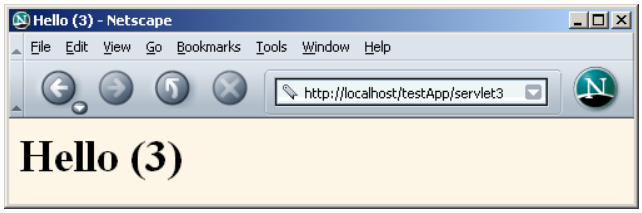

Figure 2–23 `HelloServlet3` invoked with a custom URL.