Chapter 1

Getting Started with the Java 2 SDK

Java programs come in many forms. Originally, there were just two: *applets,* which are Java programs that run in the Web browser, and *applications,* which don't. Since then, many other varieties of Java programs have joined the ranks. *Servlets* are Java programs that run in Web servers; *midlets* run on mobile information devices; and there are many more types. This chapter describes the specific differences between applets and applications and describes how to create them with the Java Software Development Kit.

Originally known as the Java Development Kit (JDK), the Java 2 Software Development Kit (SDK) is what you use to create programs for the Java 2 Platform, Standard Edition (J2SE). The SDK includes all the tools and standard Java libraries needed to create applets and applications. If you want to create server-side programs, servlets and other server-side programs, you'll need to get the Java 2 Enterprise Edition (J2EE), which works on top of the J2SE. And for developing midlets and wireless applications, there is the Java 2 Micro Edition (J2ME).

Adapted from *Mastering Java 2, J2SE 1.4*
by John Zukowski
ISBN 0-7821-4022-X $49.99

The SDK is full of command-line tools for creating and executing Java programs. You'll find a compiler to convert your Java source code into executable programs. You'll also find the means for documenting, debugging, and integrating Java programs with C/C++-based programs as well as CORBA applications. In this chapter, you'll learn how to set up your initial development environment and construct basic applets and applications. You'll get your first taste of Java source code and learn how to run programs in a Java Runtime Environment. From editing all the way through compiling and running, all these development tasks can be done for free—there are no costs involved with any of them, at least on the reference platforms supported by Sun.

NOTE

The reference platforms available from Sun are Solaris SPARC/x86, Linux x86, and Microsoft Windows. For other platforms, you will most likely need to visit the vendor of the OS and/or hardware.

Applets vs. Applications

The word *applet* usually makes one think of a "small" application. In truth, there is no such restriction on size for applets—the actual defining factor is that the applet is started by a Web browser loading an HTML file. However, since applets *are* downloaded over the Web, developers usually don't make their applets that big, in consideration of the download lag to receive the files from the Web server and then start them up in the browser.

Applications, on the other hand, are installed on the client, get started from the command line, and suffer from no download limitations. With applications, the Java version running in the browser matters not at all.

The Java Runtime Environment

Before we jump into exploring differences between applets and applications, it's important to make sure you understand the concept of the Java Runtime Environment (JRE). As its name suggests, the JRE is the environment in which your Java programs run. For applets, the browser vendor provides that environment. Though you can replace that environment with the Java Plug-in (`http://java.sun.com/products/plugin/`), for the most part you'll use what the browser vendor provides for running

Internet-deployed applets. On the other hand, for applications you can deliver the JRE with your programs, or you can rely on the user's having one already installed. The JRE is easy to replace for applications, and you'll be able to make sure your user is using a version with which you've tested your program.

The JRE is freely downloadable from `http://java.sun.com/j2se/1.4/jre/`. You don't have to have it for creating Java programs, and distribution is free when it comes time to deploy your applications.

Here's the sticky point for you readers: While this book is about developing programs for the Java 1.4 platform, the JRE version found in most Java-enabled browsers is only Java 1.1.*x*. So if you try to use any new and interesting features added to the libraries since Java 1.1, your resulting applet won't work in any version of Internet Explorer. As far as Netscape browsers go, your applet won't work in Netscape Navigator/Communicator 4.x and only has a chance of working with Netscape 6, because that browser relies on the previously mentioned Java Plug-in for its JRE, shipping with none.

NOTE

It's very important to keep in mind, though, that while the JRE found in browsers is based on Java 1.1, you *can* still run applets in browsers created from the Java 2 SDK version 1.4. You just can't use any features that weren't around during Java 1.1 time. The trick is knowing what the differences are! The easiest way to know the differences is to view the documentation for the older version of the Java runtime at `http://java.sun.com/products/jdk/1.1/docs.html`, while you're using the newer compiler.

Applets and Applications: What's the Difference?

Think of where an applet runs and where an application runs, and you'll understand nearly all the differences between them. Applets run inside browsers, and applications don't.

When it comes to using an applet, always think of downloading a program from over the Internet. Because an applet can be downloaded whenever you visit a web site, browsers must provide a tightly controlled environment to run that applet. You'll hear that area called a *sandbox*. The sandbox provides a secure environment, restricting the applet from accessing anything on the client machine and only permitting the applet

to communicate with the server it came from. In addition, the browser tells the applet when to initialize itself, when to draw itself, when to activate or deactivate itself, as well as when to unload. Lastly, the browser supplies an area within an HTML page for the applet's display, and that's the area used by the applet for drawing.

Applications, in contrast, have no such restrictions or automatic display area. Because applications run locally, they can do anything. An applet gets an area within the browser for display, but an application provides no such graphical environment by default. Yes, it can create one, but it isn't there automatically. And an applet can get input from the outside world from within its HTML page, whereas an application gets input from the command line.

So when it comes time to write a program, which one do you create? Well, it depends. There is no clear-cut answer. If you want to use any features from Java 1.4, 1.3, or even 1.2, you're pretty much stuck working with applications. However, if you don't mind installing the Java Plug-in, you can use those later versions in applets. Keep in mind that the Plug-in isn't small: It's anywhere from 9MB for Windows to over 21MB for Linux, up to nearly 30MB for 64-bit Solaris, so think about the consequences of using it. For an intranet, the Plug-in is a viable option, but for an Internet-based applet, probably not. Because corporate networks can mandate the system configuration and usually have high-speed network connections, the Plug-in may be a realistic option.

If you like the idea of Web-delivery with little to no installation cost, don't give up on applications automatically. You can use tools such as InstallShield or InstallAnywhere, which create multiplatform installable applications that can be delivered over the Web. And Java Web Start is a standard option for deploying applications but running them in a sandbox-like environment.

Working with the Java 2 SDK

When starting out to create Java programs, many developers begin with Sun's development kit, the SDK. The price is right—free—and you can always manage to work with the latest version (assuming your platform is Windows, Solaris, or Linux). The SDK is a set of command-line tools for creating Java programs, both applets and applications. All you have to do is provide an editor of your own choosing, and you have all you need.

As part of the development kit, you get the JRE and a set of libraries for your programs. The combination of the runtime environment and libraries is frequently called the *Java Platform*. With each Java release, the set of libraries changes, but the underlying runtime environment essentially does not.

The libraries that come with each Java release are essentially what differentiate Java from most other programming languages. When you know your Java version, then you know what libraries you can expect. This is true no matter what platform you are using, whether it's one of the reference implementations from Sun or a third-party version from IBM or the like. Sun has an exhaustive test suite that makes sure anything called "Java" passes its compatibility test. So if a third-party implementation doesn't pass, it isn't Java. Because your own programs are portable in this fashion, Sun has coined the term Write-Once, Run Anywhere (WORA) to describe Java programs.

The libraries available with the Java platform allow you to do almost anything. There is support for graphics, networking, data structures, XML processing, and disk I/O, among many other technologies. When you move to other types of runtime environments, such as smart cards or embedded devices instead of desktop computers, the core API set will be the same. And Sun maintains well-defined subsets of all the libraries for these other environments, so you'll know what isn't available. For instance, there is no graphical environment on smart cards, so the graphics libraries aren't available. As long as you stick to what is available across these diverse runtime environments, the same Java programs will run there, too (though usually you'll know the target runtime equipment before you start).

SDK Tools

Before we download and install the Java 2 SDK, let's take a walk through the tools that come with the kit and see how they work together. There are 23 utilities, and you'll use the following eight of them in this book:

`javac`	The Java compiler. Converts source code to byte codes.
`java`	The Java application launcher. Executes byte codes for an application.
`appletviewer`	The Java applet launcher. Executes byte codes for an applet.

`javadoc`	The Java commenting tool. Generates HTML documentation from Java source files.
`jdb`	The Java debugger. Steps through programs as they execute, sets breakpoints, and monitors behavior.
`javap`	The Java disassembler. Examines the byte codes to display information.
`jar`	Java Archive (JAR) manager. Combines and compresses multiple files and directories into one.
`javah`	Header file generator. Used when combining Java with C/C++ programs.

Assuming you have a text editor that allows you to save Java source files as raw text—that is, without the kind of formatting information that's in Microsoft Word `.doc` files—you can create and execute Java programs with these SDK tools. The process of building applications is illustrated in Figure 1.1, and Figure 1.2 illustrates the slightly different tasks involved in creating a working applet. After you download the SDK, you'll follow the steps shown in these figures to create your first Java application and applet.

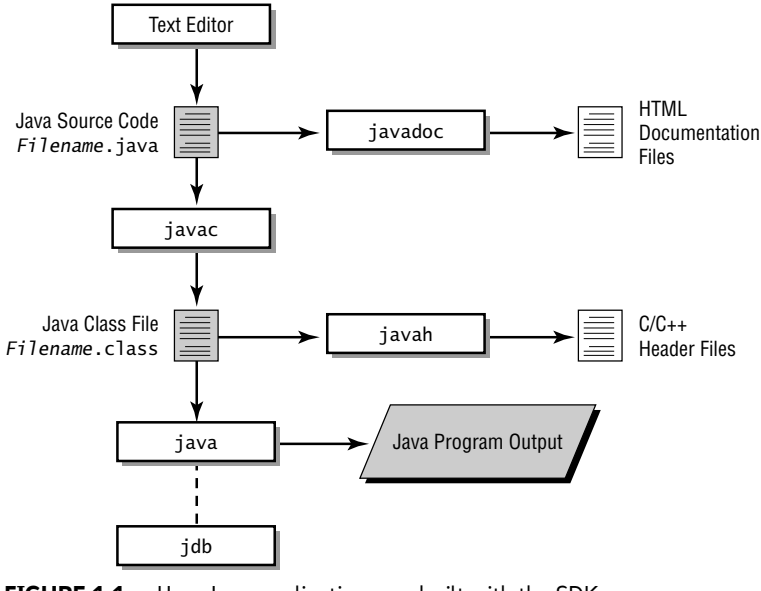

FIGURE 1.1: How Java applications are built with the SDK

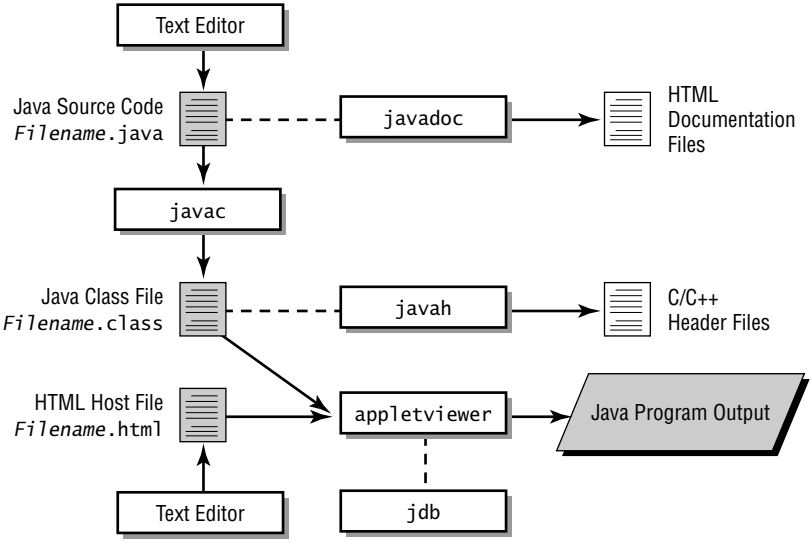

FIGURE 1.2: How Java applets are built with the SDK

Following are the remaining 15 SDK tools, listed in logical groupings:

`javaw`	Application launcher without a console. Executes byte codes for applications but runs in a windowless console.
`extcheck`	Java Archive (JAR) conflict checker. Examines JAR files for version conflicts.
`native2ascii`	Unicode converter. Translates files from one encoding format to another.
`jarsigner`, `keytool`, `policytool`	Security management tools. Allow you to create applets that break out of the sandbox.
`rmic`, `rmid`, `rmiregistry`, `serialver`	Tools for working with Remote Method Invocation (RMI).
`idlj`, `orbd`, `servertool`, `tnameserv`	Tools for working with CORBA, IDL, and IIOP.
`unregbean`	Tool to unregister Java Beans components usable over ActiveX.

Downloading and Installing the SDK

To get started working with Java, you need to download the SDK for Java 1.4. Once you install it on your machine, you'll be able to work through the examples in this book.

Downloading the SDK

You can get the 1.4 SDK for Microsoft Windows 95/98/Me/NT4/2000, Solaris SPARC/x86, or Linux directly from Sun's site at `http://java.sun.com/j2se/1.4/`. For other platforms, you'll need to visit `http://java.sun.com/cgi-bin/java-ports.cgi` to find a list of third-party ports. New Java releases are not immediately available on other platforms, so there may be some delay in availability.

Installing the SDK

The instructions for SDK installation are different for each platform. If you run into trouble, you can always follow the Installation link from the original Sun download page for help.

Windows Installation The Windows installation package comes as an InstallShield installer. Just execute the `j2sdk-1_4_0-win.exe` file and follow the prompts, and you'll get everything placed in `C:\j2sdk1.4.0` by default. You don't have to install the Native Interface Header Files, Demos, or Java Sources, but at a minimum you should install the last two.

Once the installer is done, update the system `PATH`. This is a set of directories the system uses to find programs to run. For the Java tools to work anywhere, add

```
C:\j2sdk1.4.0\bin
```

to the `PATH`. If there are multiple items in the `PATH` already, use a semicolon (`;`) as the separator.

The method of updating the system `PATH` depends on the Windows version you have. For instance, with Windows 95/98, you can add the following as the last line of the `C:\autoexec.bat` file:

```
SET PATH=C:\j2sdk1.4.0\bin;%PATH%
```

where the `%PATH%` means to keep the current path settings and simply prepend the just-installed JDK `bin` directory. For Windows NT/2000,

you use the System icon in the Control Panel to change the PATH. And Windows Me has a System Configuration Utility buried under Accessories ➢ System Tools ➢ System Information to do the same.

In most cases, after changing the PATH you'll need to reboot the machine for the changes to take effect—although you may want to wait to reboot until after you read the "Understanding CLASSPATH" section coming up.

Linux Installation Sun's Linux release runs on Intel platforms running the 2.2.12 kernel and the 2.1.2-11 or later glibc library. The officially supported Linux release is RedHat Linux 6.2. You can get either the self-extracting binary (`j2sdk-1_4_0-beta-linux-i386.bin`) or RedHat RPM file (`j2sdk-1_4_0-beta-linux-i386-rpm.bin`):

- If you download the self-extracting binary, after running the script, the SDK will be installed into the `j2sdk1.4.0` directory under the directory where you run the script. First, make the file executable with **`chmod a+x`**, and then execute. Just be sure you have write permission for whatever directory you're using.
- With the RPM file, the SDK will be installed into the `/usr/java/j2sdk1.4.0/` directory. As you do for the self-extracting binary, make the file executable and run it. However, running it only creates the `j2sdk-1.4.0.i386.rpm` file. You'll then need to run this as root with **`rpm -iv`** to install the SDK.

TIP

The RPM is not relocatable, so if you'd rather install into `/usr/local/`, don't use the RPM file.

Once the SDK is installed, add the `bin` directory under the installation directory to your PATH to make the Java tools available.

Solaris Installation For the Solaris 2.6, 7, and 8 platforms, you'll need to get the appropriate packed archive file (`j2sdk-1_4_0-beta-solsparc.sh` for Solaris/SPARC or `j2sdk-1_4_0-beta-solx86.sh` for Solaris/x86) or package bundles (`j2sdk-1_4_0-beta-solsparc.tar.Z` or `j2sdk-1_4_0-beta-solx86.tar.Z`). You may also have to download patches for your operating system. Combined, the package and the

patches may be as large as 100MB (compressed). Be sure to download and install the patches first:

- If you download the packed archive file, after running the script you'll have the SDK installed in the `j2sdk1.4.0` directory. Just make the script executable with **`chmod +x`** and then run the script from the directory in which you want `j2sdk1.4.0` created. You can always move the directory later.
- If you download the package bundles, you'll need to uncompress the `.tar.Z` file with **`zcat`**, remove the 1.3 SDK packages with **`pkgrm`** if you installed 1.3 (`SUNWj3dmo`, `SUNWj3man`, `SUNWj3dev`, and `SUNWj3rt`), and add the new 1.4 packages with **`pkgadd`**. For SPARC, the new packages are `SUNWj3rt`, `SUNWj3dev`, `SUNWj3man`, `SUNWj3dmo`, `SUNWj3rtx`, and `SUNWj3dvx`. For Intel, drop the last two. You'll wind up with the SDK installed in the `/usr/j2se` directory.

Once you've got the SDK installed, either add the `bin` directory under `j2sdk1.4.0`, or add `/usr/j2se` to your PATH so that the Java tools are available without your having to specify their full path when you need to run them.

NOTE

For convenience, all directory names in the remainder of this book will be in DOS (Windows) format, except when it's necessary to point out specific platform differences.

Understanding *CLASSPATH*

In addition to the PATH environment variable, Java relies on another variable setting: CLASSPATH. The Java runtime environment will look for user items to execute (Java classes) in the CLASSPATH environment variable. If it's not set properly, you'll run into lots of problems and go crazy pulling your hair out trying to solve them. The system does know where to look for system classes, so you have no chance of interfering with that when mucking with the CLASSPATH variable.

Under normal circumstances, you don't have to do anything at all with your CLASSPATH setting. If it's not set, the JRE looks in the current directory for user classes. If CLASSPATH is set, the JRE looks in the current

directory only if you tell it to, examining just those locations specified by the variable setting. Normally, you want the JRE to look in the current directory, because that's where it can find the classes associated with the program you're developing.

Sometimes, previously installed programs will unknowingly set your CLASSPATH for you. If they have, you'll need to manually add the current directory back to the search path. A period (`.`) in the CLASSPATH represents the current working directory, so add it to the variable.

On Windows, the CLASSPATH entries are separated by semicolons. Like PATH, setting this up depends on your platform. The new setting will look something like the following, although it depends on what else you have installed for the other entries. The important piece, in any case, is the trailing semicolon and period, as shown here:

```
SET CLASSPATH=C:\foo\bar;D:\bar\foo.jar;.
```

Under Unix (Solaris or Linux), CLASSPATH entries are separated by a colon. How exactly to set the path depends upon your shell. Here, the important piece is the colon and the period at the end:

```
setenv CLASSPATH /usr/foo:.          (For c-shell)

CLASSPATH=/usr/foo:.                 (For bourne shell)
export CLASSPATH
```

TIP

For additional information on CLASSPATH settings see the following URLs: `http://java.sun.com/j2se/1.4/docs/tooldocs/win32/classpath.html` **and** `http://java.sun.com/j2se/1.4/docs/tooldocs/solaris/classpath.html`.

Once you have everything set and have rebooted if necessary, you can run the **`java -version`** command to see if your PATH is correct and make sure you're using the appropriate version of the Java SDK. Assuming the command comes back and tells you that you're using the 1.4 release, you're all set. Otherwise, you'll have to double-check your settings. Since the `bin` directory for Java was added to the front of the PATH setting, the system should not find any other version in some other PATH location.

DOWNLOADING THE DOCUMENTATION

In addition to the SDK itself, you should download the SDK documentation from `http://java.sun.com/j2se/1.4/`. The 31MB compressed file (160MB uncompressed) includes usage and API documentation to help you become a productive developer. Though you can always look at the information online at `http://java.sun.com/j2se/1.4/docs/`, it's quicker to have the documentation set installed locally. After unpacking, be sure to bookmark the `index.html` file under the top-level `docs` directory.

Creating Java Applications

This section walks through all the steps pictured in Figure 1.1, from editing, through compilation, to execution, and beyond.

The Hello World Application

To get started with Java development, you need an editor. Unix folks will find `emacs` or `vi` handy, and Windows users can use the standard Notepad program. Pick whatever editor you're comfortable with, as long as it will save your source files as text.

A tip for Windows users: If you're using Windows Notepad as a text editor, you may find that it saves `.java` files with an added `.txt` extension—for instance, saving `Foo.java` as `Foo.java.txt`. As a short-term solution, place the filename within double quotes (`"Foo.java"`) to save. For the long term, you'll need to go into the Windows Explorer and associate `.java` files with Notepad.

For starters, you'll create a program that will print a message to the screen. This little program is typically called Hello World because that is the message displayed. Place the source code shown in Listing 1.1 in a file named `HelloWorld.java`. Java is case sensitive, so be sure to use the proper case in the filename. It has to match what is inside the source code.

Listing 1.1: The Hello World Program

```
public class HelloWorld {
  public static void main(String args[]) {
```

```
        System.out.println("Hello, World");
    }
}
```


Don't worry about fully understanding the details of the program yet. You'll learn more as you work your way through the remainder of the book. Essentially, this file contains the definition of a class named `HelloWorld`. The class contains a method named `main()`. When you try to run an application in Java, the runtime environment looks for a `main()` method. The `main()` method must have the keywords `public`, `static`, and `void` associated with it. In addition, the `main()` method must accept an array of `String` elements as argument.

NOTE

The reference implementation from Sun will run the program even if the `public` keyword is missing for `main()`. This may not work with other runtime environments, however, so just get in the habit of including the necessary keywords.

The `String args[]` part of the `main()` method declaration refers to any command-line arguments you pass into the Java program. This is similar to the C/C++ declaration

```
int main(int argc, char *argv[0]);
```

except that with C/C++, a second argument is necessary to pass in the number of elements in the array. In Java, arrays know their length, so the first argument here is unnecessary.

Aside from the case differences, Java's `main()` method declaration is practically identical to the C# mechanism to pass in command-line arguments:

```
public static void Main(string[] args)
```

The line `System.out.println("...");` in Listing 1.1 says to send the message between the quotes to `System.out`, which happens to be the console window from which you started the program.

Compiling the Application Source

Following the main path of Figure 1.1, you can see that once you've saved your program with your editor, the next step is to compile it with the compiler (`javac`). To compile your program, just enter **`javac`** followed by the case-sensitive source filename. If you haven't already, you'll need to

have a command prompt window open and be in the directory where you saved the file. So enter

```
javac HelloWorld.java
```

If the source code is error free, `javac` generates a file named `HelloWorld.class` and immediately returns to the command prompt.

When you're interested in following the inner workings of compilation, you can use the `-verbose` option with `javac`. On a single class, the results aren't too interesting, but you'll see what the compiler needs in order to verify that the program is valid. Executing this command

```
javac -verbose HelloWorld.java
```

generates the following output:

```
[parsing started HelloWorld.java]
[parsing completed 203ms]
[loading C:\J2SDK1.4.0\jre\lib\rt.jar(java/lang/Object _
  .class)]
[loading C:\J2SDK1.4.0\jre\lib\rt.jar(java/lang _
  /String.class)]
[checking HelloWorld]
[loading C:\J2SDK1.4.0\jre\lib\rt.jar(java/lang/System _
  .class)]
[loading C:\J2SDK1.4.0\jre\lib\rt.jar(java/io/PrintStream _
  .class)]
[loading C:\J2SDK1.4.0\jre\lib\rt.jar(java/io _
  /FilterOutputStream.class)]
[loading C:\J2SDK1.4.0\jre\lib\rt.jar(java/io/OutputStream _
  .class)]
[wrote HelloWorld.class]
[total 640ms]
```

If you read through your program's source code and look in the documentation for the classes, you'll find out why each of the classes is necessary. Unless specified otherwise, classes build from the `Object` class. The `main()` method of the program takes a `String` argument. Inside `main()`, there is a reference to the `System` class. The `out` variable of `System` happens to be of type `PrintStream`. The final two loaded classes have to do with `PrintStream`, a kind of `FilterOutputStream`, which is a kind of `OutputStream` (which is a kind of `Object`, but that is already loaded). With all those classes verified and no errors in the

source, the compilation is deemed to be a success, so the `.class` file is generated.

The generated `.class` file is a platform-neutral binary formal understood by the Java Virtual Machine (JVM). The JVM is what runs the byte codes in the JRE. If you are interested in learning more about the format of the file, you can read the *Java Virtual Machine Specification* at `http://java.sun.com/docs/books/vmspec/2nd-edition/html/VMSpecTOC.doc.html`.

Running the Application

After you've compiled the source code, you can run the program. Use the Java application launcher, `java`, to do this. Enter this command:

```
java HelloWorld
```

It will display the output "Hello, World" as shown in Figure 1.3.

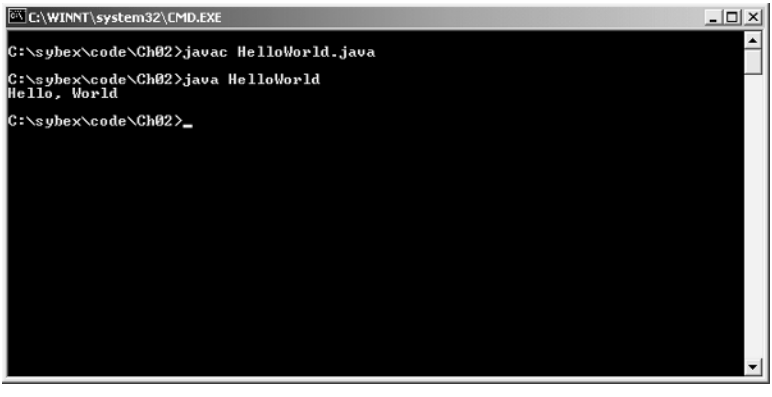

FIGURE 1.3: A Windows session to compile and execute the `HelloWorld` class

TIP

When you enter the `java` command, be sure *not* to specify the `.class` at the end of the class name to be run. Only specify the class name, not the filename.

There are some command-line options that may also be of interest when running your programs. To see a list of options, pass in `-?` as the command-line option, like this:

```
java -?
```

It will bring back a list of options you can pass into the launcher, as shown here (you won't need to worry about most of these):

```
Usage: java [-options] class [args...]
           (to execute a class)
   or  java -jar [-options] jarfile [args...]
           (to execute a jar file)

where options include:
    -hotspot      to select the "hotspot" VM
    -server       to select the "server" VM
                  If present, the option to select the VM _
                    must be first.
                  The default VM is -hotspot.

    -cp -classpath <directories and zip/jar files separated _
      by ;>
                  set search path for application classes _
                    and resources
    -D<name>=<value>
                  set a system property
    -verbose[:class|gc|jni]
                  enable verbose output
    -version      print product version and exit
    -showversion  print product version and continue
    -? -help      print this help message
    -X            print help on non-standard options
```

Listing 1.2 shows some other nonstandard options hidden under the `-X` option. These options are nonstandard and subject to change without notice. That said, some of them are actually more frequently used than the standard options. For instance, if you've created a memory-intensive program, you may find the need to increase the Java heap space with the `-Xmx` option.

Listing 1.2: Nonstandard Options for the *java* Command

```
    -Xmixed           mixed mode execution (default)
    -Xint             interpreted mode execution only
```

```
-Xbootclasspath:<directories and zip/jar files _
  separated by ;>
                set search path for bootstrap classes _
                  and resources
-Xbootclasspath/a:<directories and zip/jar files _
  separated by ;>
                append to end of bootstrap class path
-Xbootclasspath/p:<directories and zip/jar files _
  separated by ;>
                prepend in front of bootstrap class path
-Xnoclassgc     disable class garbage collection
-Xincgc         enable incremental garbage collection
-Xconcgc        enable mostly-concurrent garbage _
                  collection
-Xloggc:<file>  log GC status to a file with time stamps
-Xbatch         disable background compilation
-Xms<size>      set initial Java heap size
-Xmx<size>      set maximum Java heap size
-Xss<size>      set java thread stack size
-Xprof          output cpu profiling data
-Xrunhprof[:help]|[:<option>=<value>, ...]
                perform JVMPI heap, cpu, or monitor _
                  profiling
-Xdebug         enable remote debugging
-Xfuture        enable strictest checks, anticipating _
                  future default
-Xrs            reduce use of OS signals by Java/VM _
                  (see documentation)
```

Documenting the Application

Traveling off the beaten flowchart path a little brings us to the `javadoc` command and the important task of *commenting* code. The SDK for Java comes with a utility to enable automatic documentation of your source code. After adding special comments to your source code, you can arrange for these comments to be automatically transferred into HTML files to document your program. Any source code comments placed between `/**` and `*/` are considered *documentation comments* and are added to the generated files.

Adding a few lines of comments to the original Hello World source file, as shown in Listing 1.3, enables the testing of the utility.

Listing 1.3: Hello World with Some Comments

```
/** HelloWorld - This is the first Java program I created. */

public class HelloWorld {

/**
 * This is the initial method called by the Java launcher.
 * It will print a message to the screen.
 */

  public static void main(String args[]) {
    System.out.println("Hello, World");
  }

}
```

To generate the documentation, pass the `javadoc` command a list of source files to process (in this example, there is only one source code file):

```
javadoc HelloWorld.java
```

This will spew out a whole bunch of messages:

```
Loading source file HelloWorld.java...
Constructing Javadoc information...
Building tree for all the packages and classes...
Building index for all the packages and classes...
Generating overview-tree.html...
Generating index-all.html...
Generating deprecated-list.html...
Building index for all classes...
Generating allclasses-frame.html...
Generating index.html...
Generating packages.html...
Generating HelloWorld.html...
Generating package-list...
Generating help-doc.html...
Generating stylesheet.css...
```

All this output is essentially telling you that you have seven new HTML files and another new file for each source file passed into the command. There is also a Cascading Style Sheet (CSS) file that contains the formatting information for the HTML pages. If you open up the class-specific file, `HelloWorld.html`, you'll see the documentation comments added. This is shown in Figure 1.4.

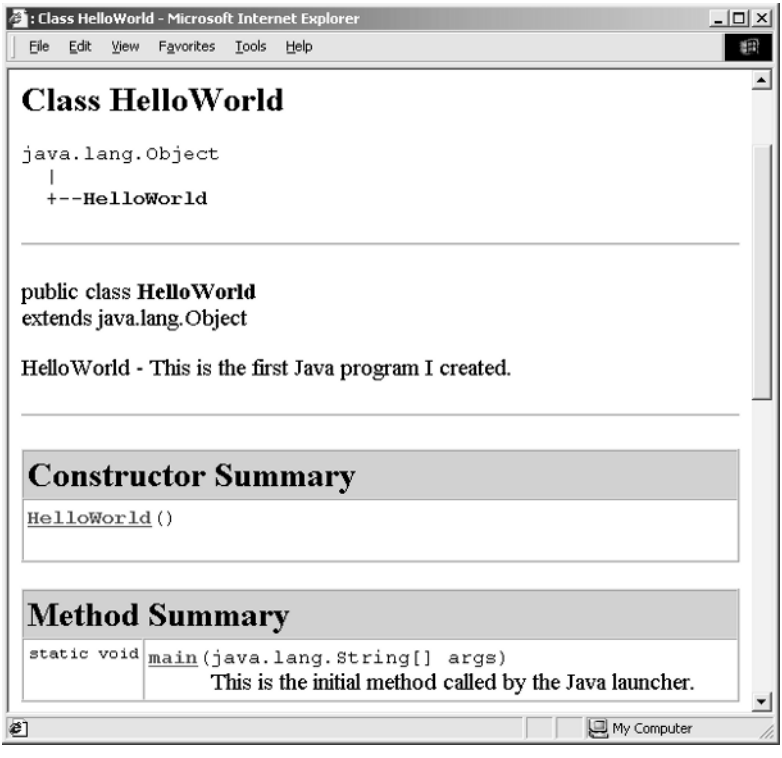

FIGURE 1.4: Viewing the `javadoc`-generated HTML file from the `HelloWorld.java` source comments

Integrating with C/C++ Code

The `javah` tool helps you to connect Java programs with C and C++ programs. This becomes necessary when there are legacy libraries around that you don't want to convert to Java but still want to use, or when you just can't do something in Java because of some platform-specific needs. As the standard libraries have grown, and as more capabilities have become standard in Java, the need to use native C/C++ source has

decreased immeasurably. If you still need to connect, the `javah` tool is around to help. Run against a class file, it will generate the necessary header files to connect to the class.

NOTE

If you find yourself in need of connecting Java and C/C++ code, check out the online tutorial at `http://java.sun.com/docs/books/tutorial/native1.1/`.

Debugging Your Application

The SDK comes with a command-line debugger, `jdb`. With `jdb`, you can execute your programs, examine variable settings, step through the source, and set breakpoints, among many other tasks. If you plan to use `jdb`, you should compile your program with the `-g` option. However, with integrated development environments such as Forté for Java (Community Edition) and JBuilder (Personal Edition) freely available, you're better off getting one of those to visually debug your programs. These helpers make the task much easier.

Disassembling Your Classes

The `javap` tool allows you to query any class and find out its list of methods (like `main()`) or find out the commands executed by the underlying JVM. You may find that you use `javap` queries frequently as an easy way to see what methods are in a class. The latter use of `javap`, to check on JVM command execution, is rare and mostly for the curious.

When `javap` is passed just the class name, like this

```
javap HelloWorld
```

it generates a list of visible methods and variables. (In this case, though, we have no variables in the `HelloWorld` class.)

```
Compiled from HelloWorld.java
public class HelloWorld extends java.lang.Object {
    public HelloWorld();
    public static void main(java.lang.String[]);
}
```

Two methods are shown here: `HelloWorld()` and `main()`. The `HelloWorld()` line is what is called a *default constructor,* a special

method that enables the creation of the class. Unless you specify otherwise, a constructor is automatically created for a class. The `main()` method is the one you created yourself.

NOTE

For more information on methods and constructors, see Chapter 2.

When you use the `-c` option with `javap`

```
javap -c HelloWorld
```

the sequence of the underlying byte codes is displayed, in addition to the prior list of methods. The `HelloWorld` class is listed just below. Essentially, these commands equate back to the source code in the `.java` file.

```
Compiled from HelloWorld.java
public class HelloWorld extends java.lang.Object {
    public HelloWorld();
    public static void main(java.lang.String[]);
}

Method HelloWorld()
   0 aload_0
   1 invokespecial #1 <Method java.lang.Object()>
   4 return

Method void main(java.lang.String[])
   0 getstatic #2 <Field java.io.PrintStream out>
   3 ldc #3 <String "Hello World">
   5 invokevirtual #4 <Method void println(java.lang.String)>
   8 return
```

Creating Java Applets

Now that you've seen how to create an application, it's time to learn how to build an applet. The tasks are almost identical, though for an applet you have to create an HTML file to load the applet. You can't just

use the Java launcher. The applet you're going to create will display a message passed in from the HTML file, or "Hello, World" if no message is supplied.

HelloWorld Applet

The process for creating the source code for the applet is identical to that for an application. Open up a text editor. For the applet, the file in the editor will be `HelloWorldApplet.java`.

NOTE

There is no language-level requirement that applet class names must end with "Applet".

The source code for the `HelloWorldApplet` class is in Listing 1.4. It looks very different from an application and will be explored shortly.

Listing 1.4: *HelloWorldApplet* Class

```
import java.awt.Graphics;
import java.applet.Applet;

/** HelloWorldApplet - This is the first Java applet I _
  created. */

public class HelloWorldApplet extends Applet {

  String msg;

/**
 * This is the initial method called by the applet launcher.
 */

  public void init() {
    msg = getParameter("message");
    if (msg == null) {
      msg = "Hello, World";
    }
  }

/**
```

```
 * This method will display a message within the browser.
 */

  public void paint(Graphics g) {
    g.drawString(msg, 20, 30);
  }
}
```

The first thing you might notice is there is no `main()` method. Applets have *life-cycle* methods, instead. Life-cycle methods are the methods of an applet that you don't call; the browser calls them for you. And they are optional, so if nothing special needs to happen, you don't have to provide the method. Life-cycle methods are listed in Table 1.1.

TABLE 1.1: Applet Life-Cycle Methods

Method	Description
`public void init()`	Initialization method. Called once when applet first loads.
`public void start()`	Startup method. Called when browser is ready to execute applet; for instance, after initialization or when user returns to a page that had the applet loaded.
`public void stop()`	Deactivation method. Called when browser is done executing applet, as when user leaves a web page.
`public void destroy()`	Termination method. Called when browser is about to unload the applet from memory.
`public void paint(Graphics g)`	Drawing method. Called when part of the applet's display area becomes invalid.

In the `init()` method of the `HelloWorldApplet`, it checks to see if a message parameter is present in the HTML loader. If it's present, the `msg` variable is set to the parameter value:

```
String msg;
. . .
msg = getParameter("message");
```

If the parameter is not set, a default message is used:

```
if (msg == null) {
  msg = "Hello, World";
}
```

The `paint()` method is even simpler. All it does is display the message on the screen. The values 20 and 30 represent screen coordinates. (You'll find more information on drawing in Chapter 13.)

```
public void paint(Graphics g) {
  g.drawString(msg, 20, 30);
}
```

The remaining lines deserve a little explanation, too. A line that begins with `import` tells the compiler where to look for a class. The `HelloWorldApplet` uses the `Graphics` and `Applet` classes:

```
import java.awt.Graphics;
import java.applet.Applet;
```

NOTE

The `HelloWorld` application used the system classes `String` and `System` and doesn't require an `import` line. These two classes are in a special location known by the compiler, so there's no need for an `import`.

All applets must "extend" from the `Applet` class. The applet runtime environment (the browser) requires this:

```
public class HelloWorldApplet extends Applet {
```

Compiling the Applet Source

The `javac` compiler doesn't care if the source file is for an applet or application. You only have to pass the filename to the compiler. However, to ensure that the generated class will execute in a browser, use the `-target 1.1` command-line option as follows:

```
javac -target 1.1 HelloWorldApplet.java
```

As long as the file is error free, you'll get `HelloWorldApplet.class` generated.

NOTE

By default, Java class files are generated for the 1.2 JRE. There were some optimization changes made to the `.class` file format after the 1.1 release.

Creating the HTML Loader

In order to run an applet, you can't just call the Java launcher, like this:

```
java HelloWorldApplet
```

If you did, you'd get an error message, because the applet doesn't include a `main()` method:

```
Exception in thread "main" java.lang.NoSuchMethodError: main
```

Since applets execute in a browser, you'll need an HTML file to load the applet. The tag used to load an applet is <APPLET>, so you'll need to place the tag in an HTML file. Any filename will do for the name of the HTML file, because there is no required mapping from `.class` file to `.html` file as there is between `.java` and `.class`.

You put the <APPLET> tag to work by setting three attributes: CODE, WIDTH, and HEIGHT. The CODE setting is the name of the applet to run—`HelloWorldApplet` in this case—and WIDTH and HEIGHT (both in pixels) are the space the browser reserves for the applet. For instance, the following code will load the applet into a 300x200 area and display the default message:

```
<APPLET CODE=HelloWorldApplet WIDTH=300 HEIGHT=200>
</APPLET>
```

NOTE

The <APPLET> tag has many more attributes. You'll learn about them and see them used as you work through this book.

Various items can occur between the opening and closing <APPLET> tags. You can put parameters there, to pass into the applet. Parameters are specified with the <PARAM> tag. They have two attributes: the NAME of the parameter and its VALUE. There is no closing </PARAM> tag. In the

HelloWorldApplet, you might pass in a different message by setting the message attribute:

```
<APPLET CODE=HelloWorldApplet WIDTH=300 HEIGHT=200>
<PARAM NAME=message VALUE="Help Me">
</APPLET>
```

Another setting that can go between the opening and closing APPLET tags is an indication of what you want displayed when Java is disabled in the browser (or just not available). In the following code, we specify the message "Java is disabled."

```
<APPLET CODE=HelloWorldApplet WIDTH=300 HEIGHT=200>
Java is disabled.
</APPLET>
```

Running the Applet

Once you have an HTML file to load the applet, you can use the appletviewer tool that comes with the SDK to load the HTML document (or you can use your browser):

```
appletviewer Hello.html
```

Once the HTML file is loaded, you'll see the applet displayed, as shown in Figure 1.5 for appletviewer and Figure 1.6 for Netscape Communicator.

FIGURE 1.5: Using appletviewer to display the HelloWorldApplet with a custom message

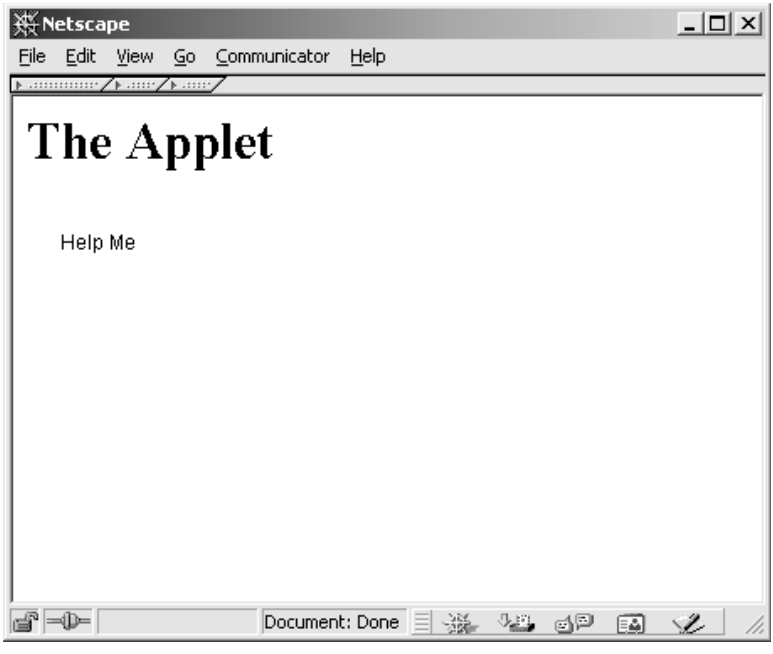

FIGURE 1.6: Using Netscape Communicator to display the `HelloWorldApplet` with a custom message

Let's compare Figure 1.5 to Figure 1.6. Notice that there happen to be some additional HTML tags in the loaded file. The `appletviewer` tool only understands the <APPLET> tag, though, so `appletviewer` will ignore the other tags. Full-blown browsers will not ignore any valid tags, however. Also, if there are multiple <APPLET> tags in a single HTML file, `appletviewer` will open each applet in its own window, whereas a browser will display them all on the same page.

Documenting the Application

Running `javadoc` on an applet is no different than for an application—just pass the source file into the tool. If you pass in multiple source files, the generated common files will include references to all files. Here's an example:

```
javadoc HelloWorld.java HelloWorldApplet.java
```

Now, if you bring up the `index.html` file, not only can you view the HTML-generated documentation for each individual class, but you also

get an index listing all the available classes. See Figure 1.7. This gets more useful as the number of classes increases and there are many interconnections.

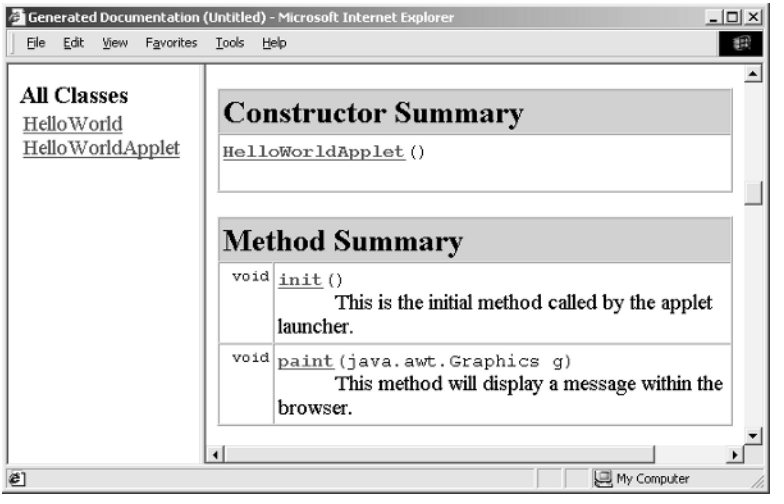

FIGURE 1.7: Viewing the `javadoc`-generated HTML file for the `HelloWorldApplet.java` source, with the combined index file

Debugging Your Applet

Debugging for applets is available with the help of the `jdb` debugger, just as for applications. To start the debugger with an applet, you need to pass the `-debug` flag to the `appletviewer` command:

```
appletviewer -debug Foo.html
```

It bears repeating here, as mentioned for applications: For serious debugging needs, you're better off with a real debugger in an IDE.

What's Next

The aspects of applications and applets described in this chapter help to form the foundation for the rest of the book. You should now have a rough idea of their differences as well as the similarities involved in working with them.

In addition to learning about applications and applets, you now have a working development environment that you can use for demonstrating and practicing the examples you'll encounter in the remaining chapters. If you prefer an integrated development environment, try out the JBuilder or Forte for Java tools available from `http://www.borland.com/jbuilder/` and `http://www.sun.com/forte/ffj/`, respectively.

In Chapter 2, be prepared to learn the basics of object-oriented programming. We've been throwing around some of its terminology already, so words like *classes* and *methods* shouldn't be totally unfamiliar to you. We'll be discussing these elements in much greater detail in the next chapter, building on the principles you've covered so far.