Getting Started

THIS chapter gives a quick introduction to the Java™ technology. First, we explain what the Java platform is and what it can do. Next are step-by-step instructions on how to compile and run two simple programs on either Win32 or UNIX/Linux platforms.[1] After that, we take a look at the code for the two programs, so you can see how they work. The chapter ends with questions and exercises to test and expand your knowledge, followed by a table of download instructions for the code used in this chapter.

The software development kits (SDKs) that Sun Microsystems provide include a minimal set of tools to let you run and compile your programs. Serious developers are advised to use a professional Integrated Development Environment (IDE).[2] See Integrated Development Environments (page 441) in the Reference Appendix for a list of IDEs.

[1] So, you're using a platform not listed here? Sun Microsystems maintains this list of third-party ports to other platforms: http://java.sun.com/cgi-bin/java-ports.cgi

[2] In fact, Java 2 SDK, Standard Edition v. 1.3, is available bundled with an IDE, the Forte™ for Java™, Community Edition. This version is included on this book's CD.

About the Java Technology

Talk about Java technology seems to be everywhere, but what exactly is it? The next two sections explain how it is both a programming language and a platform.

The Java Programming Language

The Java programming language is a high-level language that can be characterized by all of the following buzzwords:[1]

- Simple
- Object oriented
- Distributed
- Interpreted
- Robust
- Secure
- Architecture neutral
- Portable
- High performance
- Multithreaded
- Dynamic

With most programming languages, you either compile or interpret a program so that you can run it on your computer. The Java programming language is unusual in that a program is both compiled and interpreted. With the compiler, first you translate a program into an inter-

[1] Each of these terms is explained in "The Java Language Environment," a white paper by James Gosling and Henry McGilton. You can find this white paper at http://java.sun.com/docs/white/langenv/index.html

mediate language called *Java bytecodes*—the platform-independent codes interpreted by the interpreter on the Java platform. The interpreter parses and runs each Java bytecode instruction on the computer. Compilation happens just once; interpretation occurs each time the program is executed. Figure 1 illustrates how this works.

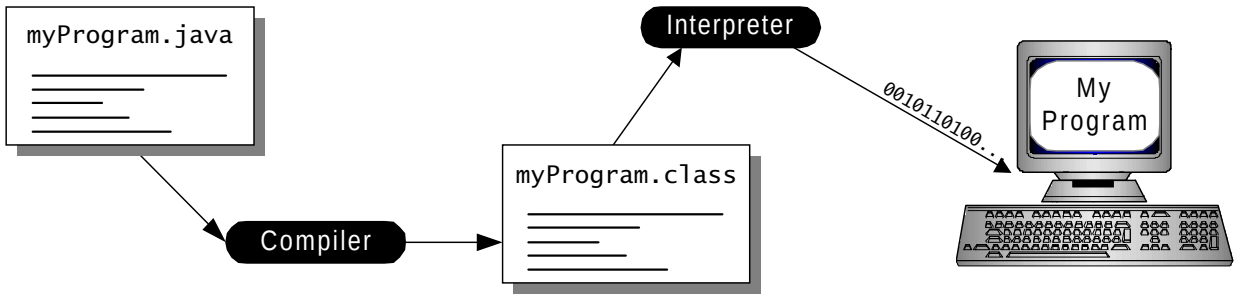

Figure 1 Programs written in the Java programming language are first compiled and then interpreted.

You can think of Java bytecodes as the machine code instructions for the *Java Virtual Machine* (Java VM). Every Java interpreter, whether it's a development tool or a Web browser that can run applets, is an implementation of the Java VM.

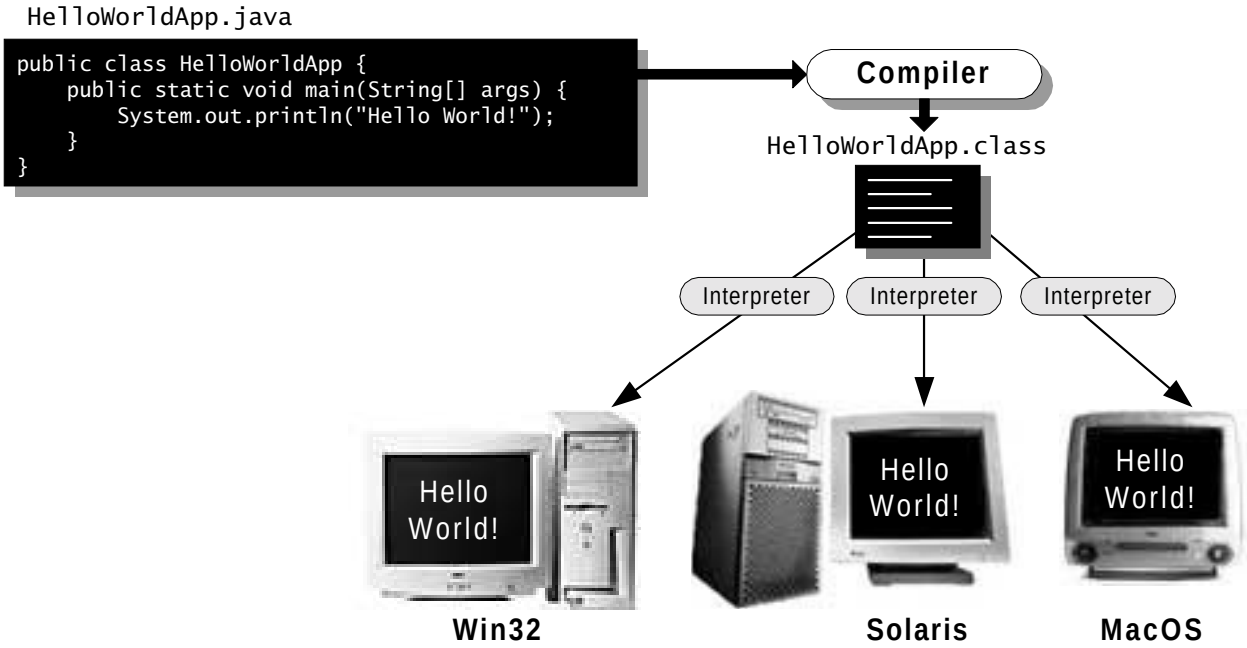

Figure 2 Programs can be written once and run on almost any platform.

Java bytecodes help make “write once, run anywhere” possible. You can compile your program into bytecodes on any platform that has a Java compiler. The bytecodes can then be run on any implementation of the Java VM. That means that as long as a computer has a Java VM, the same program written in the Java programming language can run on Windows 2000, a Solaris workstation, or on an iMac, as shown in Figure 2.

The Java Platform

A *platform* is the hardware or software environment in which a program runs. We’ve already mentioned some of the most popular platforms like Windows 2000, Linux, Solaris, and MacOS. Most platforms can be described as a combination of the operating system and hardware. The Java platform differs from most other platforms in that it’s a software-only platform that runs on top of other hardware-based platforms.

The Java platform has two components:

- The *Java Virtual Machine* (Java VM)
- The *Java Application Programming Interface* (Java API)

You’ve already been introduced to the Java VM. It’s the base for the Java platform and is ported onto various hardware-based platforms.

The Java API is a large collection of ready-made software components that provide many useful capabilities, such as graphical user interface (GUI) widgets. The Java API is grouped into libraries of related classes and interfaces; these libraries are known as *packages*. The next section highlights what functionality some of the packages in the Java API provide.

Figure 3 depicts a program that’s running on the Java platform. As the figure shows, the Java API and the virtual machine insulate the program from the hardware.

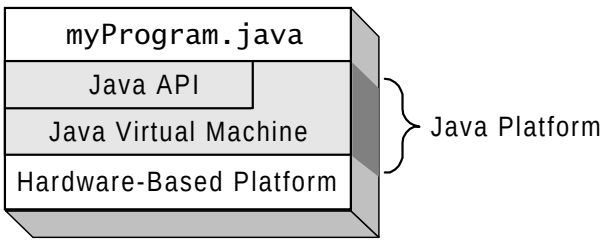

Figure 3 The Java API and the virtual machine insulate the program from hardware dependencies.

Native code is code that after you compile it, the compiled code runs on a specific hardware platform. As a platform-independent environment, the Java platform can be a bit slower than native code. However, smart compilers, well-tuned interpreters, and just-in-time bytecode compilers can bring performance close to that of native code without threatening portability.

What Can Java Technology Do?

The most common types of programs written in the Java programming language are *applets* and *applications*. If you've surfed the Web, you're probably already familiar with applets. An applet is a program that adheres to certain conventions that allow it to run within a Java-enabled browser. To see a running applet, go to this page in the online version of this tutorial:

```
http://java.sun.com/docs/books/tutorial/getStarted/index.html
```

There you can see an animation of the Java platform's mascot, Duke, waving at you:

However, the Java programming language is not just for writing cute, entertaining applets for the Web. The general-purpose, high-level Java programming language is also a powerful software platform. Using the generous API, you can write many types of programs.

An application is a standalone program that runs directly on the Java platform. A special kind of application known as a *server* serves and supports clients on a network. Examples of servers are Web servers, mail servers, and print servers.

Another specialized program is a *servlet*. A servlet can almost be thought of as an applet that runs on the server side. Java Servlets are a popular choice for building interactive web applications, replacing the use of CGI scripts. Servlets are similar to applets in that they are runtime extensions of applications. Instead of working in browsers, though, servlets run within Java Web servers, configuring or tailoring the server.

How does the API support all these kinds of programs? It does so with packages of software components that provide a wide range of functionality. Every full implementation of the Java platform gives you the following features:

- **The essentials:** Objects, strings, threads, numbers, input and output, data structures, system properties, date and time, and so on.
- **Applets:** The set of conventions used by Java applets.
- **Networking:** URLs, TCP (Transmission Control Protocol), UDP (User Datagram Protocol) sockets, and IP (Internet Protocol) addresses.
- **Internationalization:** Help for writing programs that can be localized for users worldwide. Programs can automatically adapt to specific locales and be displayed in the appropriate language.

- **Security:** Both low level and high level, including electronic signatures, public and private key management, access control, and certificates.
- **Software components:** Known as JavaBeans™, can plug into existing component architectures.
- **Object serialization:** Allows lightweight persistence and communication via RMI (Remote Method Invocation).
- **Java Database Connectivity (JDBC™):** Provides uniform access to a wide range of relational databases.

The Java platform also has APIs for 2D and 3D graphics, accessibility, servers, collaboration, telephony, speech, animation, and more. Figure 4 depicts what is included in the Java 2 SDK.

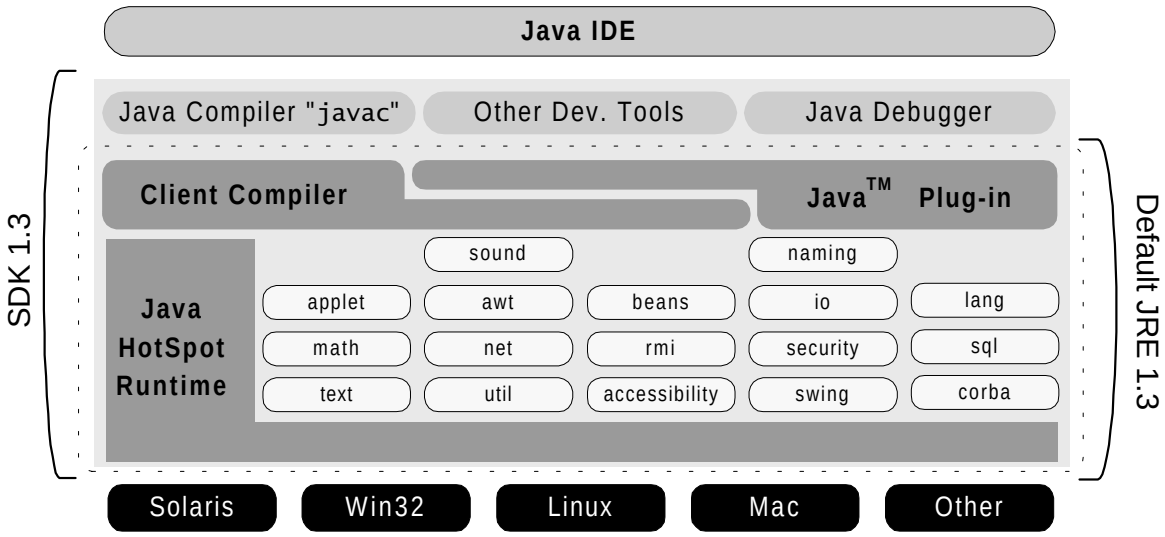

Figure 4 The Java 2 SDK, Standard Edition v. 1.3. The Java 2 Runtime Environment (JRE) consists of the virtual machine, the Java platform core classes, and supporting files. The Java 2 SDK includes the JRE and development tools such as compilers and debuggers.

This book covers the Java programming language and parts of the core API that beginning- to intermediate-level programmers will use most frequently. If you need additional information not found in this book, you can explore the other two books in *The Java Tutorial* series: *The JFC Swing Tutorial* and *The Java Tutorial Continued.* The contents of both books are included on the CD that accompanies this book and can be found in the online tutorial:

```
http://java.sun.com/docs/books/tutorial/index.html
```

How Will Java Technology Change My Life?

We can't promise you fame, fortune, or even a job if you learn the Java programming language. But it *is* likely to make your programs better, and it requires less effort than do other languages. We believe that the Java programming language will help you do the following:

- **Get started quickly:** Although the Java programming language is a powerful object-oriented language, it's easy to learn, especially for programmers already familiar with C or C++.
- **Write less code:** Comparisons of program metrics (class counts, method counts, and so on) suggest that a program written in the Java programming language can be four times smaller than the same program in C++.
- **Write better code:** The Java programming language encourages good coding practices, and its garbage collection helps you avoid memory leaks. Its object orientation, its JavaBeans component architecture, and its wide-ranging, extensible API let you reuse other people's code and introduce fewer bugs.
- **Develop programs more quickly:** Your development time may be twice as fast as writing the same program in C++. Why? You write fewer lines of code with the Java programming language, and it is a simpler programming language than C++.
- **Avoid platform dependencies with 100% Pure Java™:** You can keep your program portable by avoiding the use of libraries written in other languages. The 100% Pure Java™ Product Certification Program has a repository of historic process manuals, white papers, brochures, and similar materials online at: `http://java.sun.com/100percent/`
- **Write once, run anywhere:** Because 100% Pure Java programs are compiled into machine-independent bytecodes, they run consistently on any Java platform.
- **Distribute software more easily:** You can upgrade certain types of programs such as applets easily from a central server. Applets take advantage of the feature of allowing new classes to be loaded "on the fly," without recompiling the entire program.

Let's get started learning the Java programming language with a simple program, "Hello World." Depending on which platform you are using, you will want to read one of the next three sections: First Steps (Win32) (page 8) gives detailed instructions on compiling your first program on the Windows platform, First Steps (UNIX/Linux) (page 16) has instructions for the UNIX and Linux platforms, and First Steps (MacOS) (page 24) covers the MacOS platforms. Then don't miss the A Closer Look at HelloWorld (page 32) section which explains some basic features of the Java programming language as demonstrated in the `HelloWorld` program.

First Steps (Win32)

The following detailed instructions will help you write your first program. These instructions are for users on Win32 platforms, which include Windows 95/98 and Windows NT/2000. (UNIX and Linux instructions are on page 16. Users on MacOS platforms can find instructions on page 24.) We start with a checklist of what you need to write your first program. Next we cover the steps to creating an application, steps to creating an applet, and explanations of error messages you may encounter.

A Checklist

To write your first program, you need:

1. **The Java 2 SDK, Standard Edition:** The Java 2 SDK software is included on the CD that accompanies this book. You can download this platform to your PC or check `http://java.sun.com/products/` for the latest version.[1]
2. **A text editor:** In this example, we'll use NotePad, the simple editor included with the Windows platforms. To find NotePad, go to the Start menu and select `Programs` > `Accessories` > `NotePad`. You can easily adapt these instructions if you use a different text editor.

Note: You may want to consider using an IDE to help you write your programs. Java 2 SDK, Standard Edition v. 1.3, is available bundled with an IDE, the Forte™ for Java™, Community Edition. This version is included on this book's CD.

Creating Your First Application

Your first program, `HelloWorldApp`, will simply display the greeting "Hello world!" To create this program, you complete each of the following steps.

- **Create a source file.** A source file contains text, written in the Java programming language, that you and other programmers can understand. You can use any text editor to create and to edit source files.

---

[1] Before version 1.2, the software development kit provided by Sun Microsystems was called the "JDK."

- **Compile the source file into a bytecode file.** The compiler takes your source file and translates the text into instructions that the Java virtual machine can understand. The compiler converts these instructions into a bytecode file.
- **Run the program contained in the bytecode file.** The Java interpreter installed on your computer implements the Java VM. This interpreter takes your bytecode file and carries out the instructions by translating them into instructions that your computer can understand.

Create a Source File

To create a source file, you have two options. You can save the file `HelloWorldApp.java`[1] on your computer and avoid a lot of typing. Then you can go straight to the second step of compiling the source file (page 10). Or, you can follow these longer instructions.

First, start NotePad. In a new document, type in the following code.

```
/**
 * The HelloWorldApp class implements an application that
 * displays "Hello World!" to the standard output.
 */
public class HelloWorldApp {
    public static void main(String[] args) {
        // Display "Hello World!"
        System.out.println("Hello World!");
    }
}
```

Be Careful When You Type: Type all code, commands, and file names exactly as shown. The compiler and interpreter are *case sensitive*, so you must capitalize consistently. In other words, `HelloWorldApp` is not equivalent to `helloworldapp`.

Second, save this code to a file. From the menu bar, select `File` > `Save As`. In the `Save As` dialog, do the following.

- Using the `Save in` drop-down menu, specify the folder (directory) where you'll save your file. In this example, the folder is `java` on the C drive.
- In the `File name` text box, type `"HelloWorldApp.java"`, including the double quotation marks. The quotation marks force the file to be saved as a `.java` file rather than as a `".txt"` text file.
- From the `Save as type` drop-down menu, choose `Text Document`.

---

[1] Throughout this book we use the CD icon to indicate that the code (in this case, `HelloWorldApp.java`) is available on the CD and online. See Code Samples (page 43) for the file location on the CD and online.

When you're finished, the dialog box should look like Figur e5.

Figure 5 Saving the `HelloWorldApp.java` file with the correct `.java` extension.

Now click `Save` and exit NotePad.

Compile the Source File

From the `Start` menu, select the `MS-DOS Prompt` (Windows 95/98) or `Command Prompt` (Windows NT/2000) application. When the application launches, it should look like Figure 6.

Figure 6 The prompt in the MS-DOS Prompt application.

The prompt shows your *current directory*. When you bring up the prompt for Windows 95/98, your current directory is usually `WINDOWS` on your C drive (as shown in Figure 6) or `WINNT` for Windows NT. To compile your source code file, change your current directory to the one in which your file is located. For example, if your source directory is `java` on the C drive, you would type the following command at the prompt and press `Enter`:

```
cd c:\java
```

Now the prompt should change to `C:\java`. To change to a directory on a different drive, you must type an extra command. As shown Figure7, to change to the `java` directory on the D drive, you must reenter the drive, `d:`.

```
C:\WINDOWS>cd d:\java

C:\WINDOWS>d:

D:\java>
```

Figure 7 Changing to a directory on another drive requires an extra command—you must reenter the drive letter followed by a colon.

In our example, the `java` directory is on the C drive. The `dir` command lists the files in your current directory. If your current directory is `C:\java` and you enter `dir` at the prompt, you should see your file (Figure 8).

Figure 8 `HelloWorldApp.java` listed in the current directory.

Now you can compile. At the prompt, type the following command and press `Enter`:

```
javac HelloWorldApp.java
```

If your prompt reappears without error messages, congratulations. You have successfully compiled your program. If you encounter errors, see Error Explanations (Win32) (page 14) to help you fix the problems.

The compiler has generated a Java bytecode file, `HelloWorldApp.class`. At the prompt, type `dir` to see the new file that was generated (Figure 9).

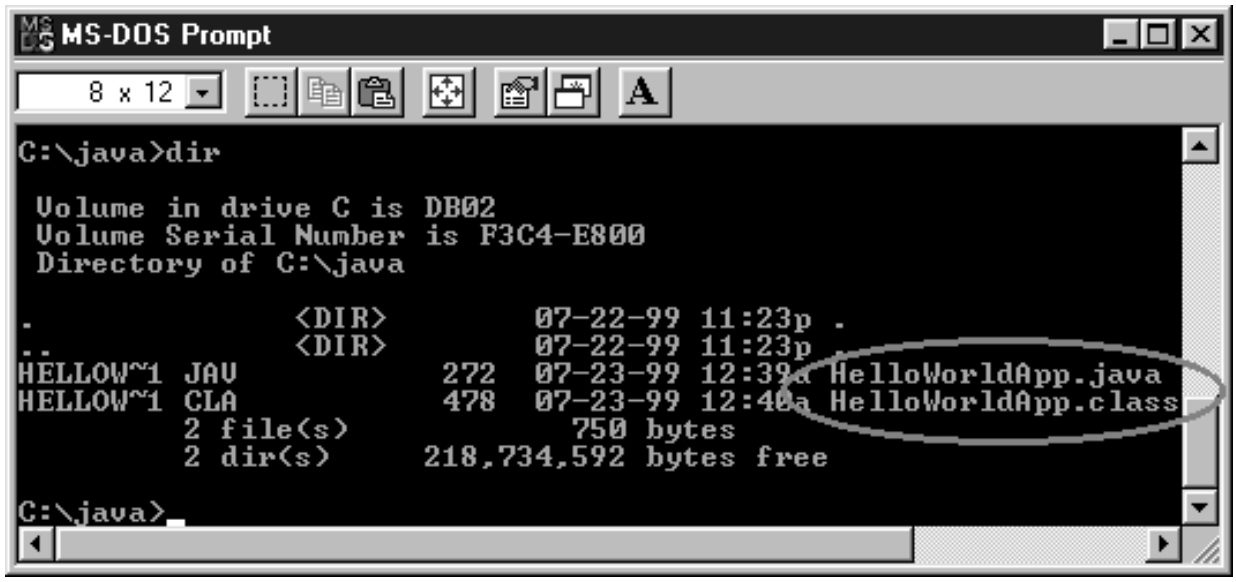

Figure 9 After you compile `HelloWorld.java`, the bytecode file, `HelloWorldApp.class`, is created in the same directory.

Now that you have a `.class` file, you can run your program.

Run the Program

In the same directory, enter the following command at the prompt:

```
java HelloWorldApp
```

Figure 10 shows what you should see.

Figure 10 When you run your `HelloWorldApp` program, you should see this result.

Creating Your First Applet

`HelloWorldApp` is an example of an *application*, a standalone program. Now you will create an *applet* called `HelloWorld`, which also displays the greeting "Hello world!". Unlike `HelloWorldApp`, however, the applet runs in a Java-enabled Web browser, such as the HotJava™ browser, Netscape Navigator, or Microsoft Internet Explorer.

To create this applet, you'll perform the basic steps as before: create a source file, compile the source file, and run the program. However, unlike for an application, you must also create an HTML file.

Create a Source File

You have two options to create a source file. You can save the files `HelloWorld.java` and `Hello.html`[1] on your computer and avoid a lot of typing. Then you can go straight to the second step of compiling the source file (page 14). Or you can follow these instructions.

First, start NotePad. In a new document, type the following code:

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

public class HelloWorld extends Applet {
    public void paint(Graphics g) {
        // Display "Hello World!"
        g.drawString("Hello world!", 50, 25);
    }
}
```

Save this code to a file called `HelloWorld.java`.

Second, you also need an HTML file to accompany your applet. Type the following code into a new NotePad document:

```
<HTML>
    <HEAD>
        <TITLE>A Simple Program</TITLE>
    </HEAD>
    <BODY>
        Here is the output of my program:
       <APPLET CODE="HelloWorld.class" WIDTH=150 HEIGHT=25>
       </APPLET>
    </BODY>
</HTML>
```

---

[1] `HelloWorld.java` and `Hello.html` are available on this book's CD and online. See Code Samples (page 43).

Save this code to a file called `Hello.html`.

Compile the Source File

At the prompt, type the following command and press `Return`:

```
javac HelloWorld.java
```

The compiler should generate a Java bytecode file, `HelloWorld.class`.

Run the Program

Although you can use a Web browser to view your applets, you may find it easier to test your applets by using the simple `appletviewer` application that comes with the Java platform. To view the `HelloWorld` applet using `appletviewer`, enter at the prompt:

```
appletviewer Hello.html
```

Figure 11 shows what you should see.

Figure 11 The successful execution of the `HelloWorld` applet.

Congratulations! Your applet works. If you encounter errors, see Common Problems and Their Solutions (page 351) in the Appendix to help you fix the problems.

Error Explanations (Win32)

Here we list the most common errors users have when compiling and running their first application or applet using the Java 2 SDK or an earlier JDK. For more error explanations, consult the section Common Problems and Their Solutions (page 351) in Appendix A.

`Bad command or file name` *(Windows 95/98)*

`The name specified is not recognized as an internal or external command, operable program or batch file` *(Windows NT/2000)*

If you receive this error, Windows cannot find the Java compiler, `javac`.

Here's one way to tell Windows where to find `javac`. Suppose that you installed the Java 2 Software Development Kit in `C:\jdk1.3`. At the prompt, you would type the following command and press `Enter`:

```
C:\jdk1.3\bin javac HelloWorldApp.java
```

Note: If you choose this option, each time you compile or run a program, you must precede your `javac` and `java` commands with `c:\jdk1.3\bin` or the directory where you saved the Java 2 SDK, followed by `\bin`. The `bin` directory contains the compiler and interpreter. To avoid this extra typing, consult the section Update the PATH Variable (Win32) (page 443) in the Appendix.

`Exception in thread "main" java.lang.NoClassDefFoundError: HelloWorldApp`

If you receive this error, the interpreter cannot find your bytecode file, `HelloWorldApp.class`.

One of the places `java` tries to find your bytecode file is your current directory. So, if your bytecode file is in C, you should change your current directory to that. To change your directory, type the following command at the prompt and press `Enter`:

```
cd c:
```

The prompt should change to `C:`. If you enter `dir` at the prompt, you should see your `.java` and `.class` files. Now enter `java HelloWorldApp` again.

If you still have problems, you might have to change your `CLASSPATH` variable. To see whether this is necessary, try clobbering the class path with the following command:

```
set CLASSPATH=
```

Now enter `java HelloWorldApp` again. If the program works now, you'll have to change your `CLASSPATH` variable. For more information, consult the section Classpath Help (page 442).

First Steps (UNIX/Linux)

These instructions tell you how to compile and run your first programs on UNIX and Linux platforms. (Win32 instructions are on page 16. Users on MacOS platforms can find instructions on page 24.) We start with a checklist of what you need to write your first program. Next we cover the steps to creating an application, steps to creating an applet, and explanations of error messages you may encounter.

A Checklist

To write your first program, you will need:

1. **The Java 2 SDK, Standard Edition:** The Java 2 SDK software is included on the CD that accompanies this book. You can download this SDK to your workstation or check `http://java.sun.com/products/` for the latest version.[1]
2. **A text editor:** In this example, we'll use Pico, an editor available on many UNIX-based platforms. You can easily adapt these instructions if you use a different text editor, such as vi or emacs.

These two items are all you need to write your first program.

Creating Your First Application

Your first program, `HelloWorldApp`, will simply display the greeting "Hello world!" To create this program, you will complete each of the following steps.

- **Create a source file.** A source file contains text, written in the Java programming language, that you and other programmers can understand. You can use any text editor to create and to edit source files.
- **Compile the source file into a bytecode file.** The compiler, `javac`, takes your source file and translates the text into instructions that the Java Virtual Machine (JVM) can understand. The compiler converts these instructions into a bytecode file.
- **Run the program contained in the bytecode file.** The Java interpreter installed on your computer implements the Java VM. This interpreter takes your bytecode file and carries out the instructions by translating them into instructions that your computer can understand.

[1] The Linux platform was first supported in the Java 2 SDK, Standard Edition v 1.3 release. Before version 1.2, the software development kit provided by Sun Microsystems was called the "JDK."

Create a Source File

You have two options. You can save the file `HelloWorldApp.java`[1] on your computer and avoid a lot of typing. Then you can go straight to the second step of compiling the file (page 18). Or you can follow these (longer) instructions.

First, open a shell, or "terminal," window (Figure 12).

Figure 12 A UNIX terminal window.

When you first bring up the prompt, your current directory will usually be your home directory. You can change your current directory to your home directory at any time by typing `cd` at the prompt and then pressing `Return`.

We recommend that you keep the files you create in a separate directory. You can create a directory by using the command `mkdir`. For example, to create the directory `java` in your home directory, you would first change your current directory to your home directory by entering the following command:

```
cd
```

Then you would enter the following command:

```
mkdir java
```

To change your current directory to this new directory, you would then enter:

```
cd java
```

Now you can start creating your source file. Start the Pico editor by typing `pico` at the prompt and pressing `Return`. If the system responds with the message **`pico: command not`**

---

[1] Throughout this book we use the CD icon to indicate that the code (in this case, `HelloWorldApp.java`) is available on the CD and online. See Code Samples (page 43) for the file location on the CD and online.

found, Pico is probably unavailable. Consult your system administrator for more information, or use another editor.

When you start Pico, it will display a new, blank buffer. This is the area in which you will type your code.

Type the following code into the new buffer:

```
/**
 * The HelloWorldApp class implements an application that
 * displays "Hello World!" to the standard output.
 */
public class HelloWorldApp {
    public static void main(String[] args) {
        // Display "Hello World!"
        System.out.println("Hello World!");
    }
}
```

Be Careful When You Type: Type all code, commands, and file names exactly as shown. The compiler and interpreter are *case sensitive*, so you must capitalize consistently. In other words, `HelloWorldApp` is not equivalent to `helloworldapp`.

Save the code with the name `HelloWorldApp.java` by typing `Ctrl-O` in your Pico editor. On the bottom line of your editor, you will see the prompt `File Name to write`. Enter `HelloWorldApp.java`, preceded by the directory where you want to create the file. For example, if `/home/myname/` is your home directory and you want to save `HelloWorldApp.java` in the directory `/home/myname/java`, you would type `/home/myname/java/HelloWorldApp.java` and press `Return`. Type `Ctrl-X` to exit Pico.

Compile the Source File

Bring up another shell window. To compile your source file, change your current directory to the one in which your file is located. For example, if your source directory is `/home/myname/java`, you would type the following command at the prompt and press `Return`:[1]

```
cd /home/myname/java
```

You can type `pwd` at the prompt to see your current directory. In this example the current directory has been changed to `/home/myname/java`. If you enter `ls` at the prompt, you should see your file listed.

---

[1] You could also change to the source directory with two commands: Type `cd`, press `Return`, then type `java` and press `Return`. Typing `cd` alone changes you to your home directory (i.e., to `/home/myname`).

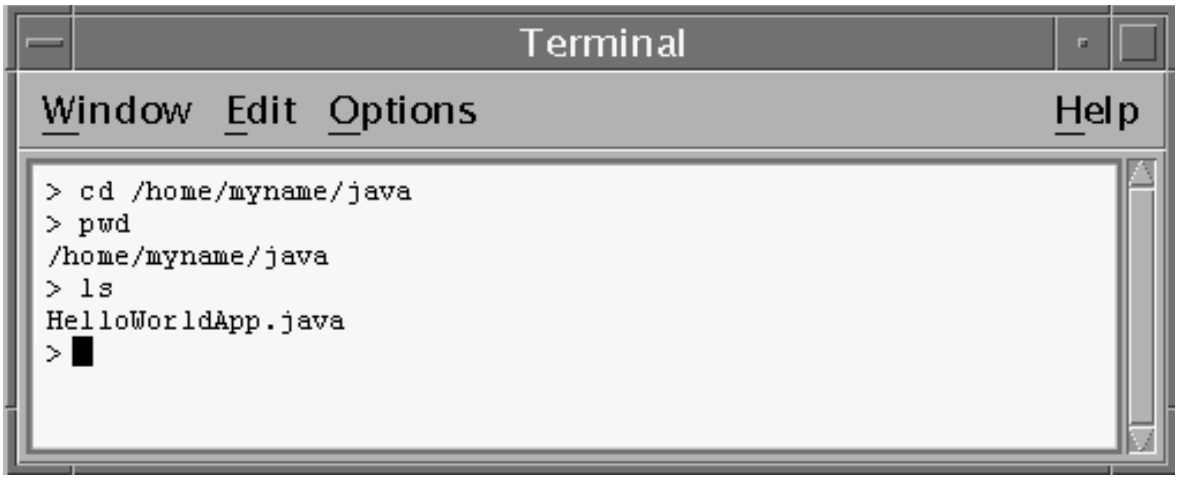

Figure 13 The `HelloWorldApp.java` file listed in the current directory.

Now you can compile. At the prompt, type the following command and press `Return`:

```
javac HelloWorldApp.java
```

If your prompt reappears without error messages, congratulations. You successfully compiled your program. If you encounter errors, see Common Problems and Their Solutions (page 351) in the Appendix to help you fix the problems.

The compiler has generated a bytecode file, `HelloWorldApp.class`. At the prompt, type `ls` to see the new file.

Figure 14 Compiling `HelloWorld.java` creates the bytecode file, `HelloWorldApp.class`, in the same directory.

Now that you have a `.class` file, you can run your program.

Run the Program

In the same directory, enter at the prompt: `java HelloWorldApp`. Figure 15 shows what you should see.

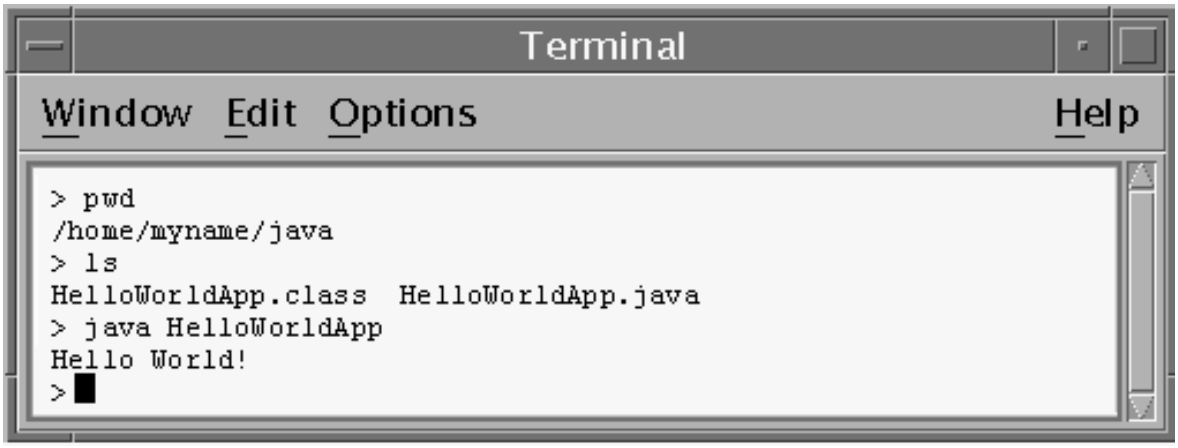

Figure 15 Running the `HelloWorldApp` program.

Creating Your First Applet

`HelloWorldApp` is an example of an application, a standalone program. Now you will create an applet called `HelloWorld`, which also displays the greeting "Hello world!". Unlike `HelloWorldApp`, however, the applet runs in a Java-enabled Web browser, such as HotJava, Netscape Navigator, or Microsoft Internet Explorer.

To create this applet, you'll perform the basic steps as before: create a source file, compile the source file, and run the program. However, unlike for an application, you must also create an HTML file.

Create a Source File

Again, you have two options. You can save the files `HelloWorld.java` and `Hello.html`[1] on your computer and avoid a lot of typing. Then you can go straight to Compile the Source File (page 21). Or you can follow these instructions.

First, start Pico. Type the following code into a new buffer:

[1] `HelloWorld.java` and `Hello.html` are available on this book's CD and online. See Code Samples (page 43).

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

public class HelloWorld extends Applet {
    public void paint(Graphics g) {
        // Display "Hello World!"
        g.drawString("Hello world!", 50, 25);
    }
}
```

Save this code to a file named `HelloWorld.java`. Type `Ctrl-X` to exit Pico.

Second, you also need an HTML file to accompany your applet. Restart Pico and type the following code into a new buffer:

```
<HTML>
    <HEAD>
        <TITLE>A Simple Program</TITLE>
    </HEAD>
    <BODY>
        Here is the output of my program:
       <APPLET CODE="HelloWorld.class" WIDTH=150 HEIGHT=25>
       </APPLET>
    </BODY>
</HTML>
```

Save this code to a file called `Hello.html` in the same directory as your `.java` file.

Compile the Source File

At the prompt, type the following command and press Return:

```
javac HelloWorld.java
```

The compiler should generate a Java bytecode file, `HelloWorld.class`.

Run the Program

Although you can use a Web browser to view your applets, you may find it easier to test your applets by using the simple `appletviewer` application that comes with the Java 2 SDK. To view the `HelloWorld` applet using `appletviewer`, enter at the prompt:

```
appletviewer Hello.html
```

Figure 16 shows what you should see.

Figure 16 The successful execution of the `HelloWorld` applet.

Congratulations! Your applet works. If you encounter errors, see Common Problems and Their Solutions (page 351) in the Appendix to help you fix the problems.

Error Explanations (UNIX/Linux)

Here we list the most common errors users have when compiling and running their first application or applet. For more error explanations, consult the section Common Problems and Their Solutions (page 351) in Appendix A.

`javac: Command not found`

If you receive this error, the operating system cannot find the Java compiler, `javac`. Here's one way to tell it where to find `javac`. Suppose that you installed the Java 2 Software Development Kit in `/usr/local/jdk1.3`. At the prompt, you would type the following command and press Return:

```
/usr/local/jdk1.3/bin/javac HelloWorldApp.java
```

Note: If you choose this option, each time you compile or run a program, you must precede your `javac` and `java` commands with `/usr/local/jdk1.3/bin`. To avoid this extra typing, consult the section Update the PATH Variable (UNIX) (page 445) in the Appendix.

`Exception in thread "main" java.lang.NoClassDefFoundError: HelloWorldApp`

If you receive this error, the interpreter cannot find your bytecode file, `HelloWorldApp.class`. One of the places the interpreter tries to find your bytecode file is your current directory. So, if your bytecode file is in `/home/myname/java/`, you should

change your current directory to that directory. To change your directory, type the following command at the prompt and press Return:

```
cd /home/myname/java
```

Type `pwd` at the prompt; you should see `/home/myname/java`. If you type `ls` at the prompt, you should see your `.java` and `.class` files. Now enter `java HelloWorldApp` again.

If you still have problems, you might have to change your `CLASSPATH` variable. To see whether this is necessary, try "clobbering" the class path with the following command:

```
set CLASSPATH=
```

Now enter `java HelloWorldApp` again. If the program works now, you'll have to change your `CLASSPATH` variable. For more information, consult the section Classpath Help (page 442).

First Steps (MacOS)

The following detailed instructions will help you write your first program. These instructions are for users on MacOS platforms. (Users on Win32 platforms can find instructions on page 8. UNIX and Linux instructions are on page16.) We start with a checklist of what you need to write your first program. Next we cover the steps to creating an application and steps to creating an applet.

A Checklist

To write your first program, you need:

1. A development environment for the Java platform. You can download the Macintosh Runtime Environment for Java Software Development Kit (MRJ SDK) from Apple's Web site at this address: `http://developer.apple.com/java/text/download.html`
2. A runtime environment for the same version of the Java platform. You can download the Macintosh Runtime Enviroment for Java (MRJ) from Apple's Web site. `http://developer.apple.com/java/text/download.html`
3. Stuffit Expander 5.5 to open these files. You can download this program from Aladdin Systems' Web site at this address: `http://www.aladdinsys.com/expander/expander_mac_login.html`
4. **A text editor:** In this example, we'll use SimpleText, the basic text editor included with the Mac OS platforms. To find SimpleText, from the File menu select `Find`, type `SimpleText` and press the `Find` button. You can easily adapt these instructions if you use a different text editor.

Creating Your First Application

Your first program, `HelloWorldApp`, will simply display the greeting "Hello world!" To create this program, you complete each of the following steps.

- **Create a source file.** A source file contains text, written in the Java programming language, that you and other programmers can understand. You can use any text editor to create and to edit source files.

- **Compile the source file into a bytecode file.** The compiler takes your source file and translates the text into instructions that the Java virtual machine can understand. The compiler converts these instructions into a bytecode file.
- **Run the program contained in the bytecode file.** The Java interpreter installed on your computer implements the Java VM. This interpreter takes your bytecode file and carries out the instructions by translating them into instructions that your computer can understand.

Create a Source File

To create a source file, you have two options. You can save the file `HelloWorldApp.java`[1] on your computer and avoid a lot of typing. Then you can go straight to the second step of compiling the source file (page 26). Or, you can follow these longer instructions.

First, start SimpleText. In a new document, type in the following code.

```
/**
 * The HelloWorldApp class implements an application that
 * displays "Hello World!" to the standard output.
 */
public class HelloWorldApp {
    public static void main(String[] args) {
        // Display "Hello World!"
        System.out.println("Hello World!");
    }
}
```

Be Careful When You Type: Type all code, commands, and file names exactly as shown. The compiler and interpreter are *case sensitive*, so you must capitalize consistently. In other words, `HelloWorldApp` is not equivalent to `helloworldapp`.

Second, save this code to a file. From the menu bar, select `File` > `Save As`. In the `Save As` dialog, do the following.

- Specify the folder where you'll save your file. In this example, the folder is called `MRJ SDK 2.2`.
- In the `Save this document as:` text box, type `HelloWorldApp.java`.

[1] Throughout this book we use the CD icon to indicate that the code (in this case, `HelloWorldApp.java`) is available on the CD and online. See Code Samples (page 43) for the file location on the CD and online.

When you're finished, the dialog box should look like Figure 17.

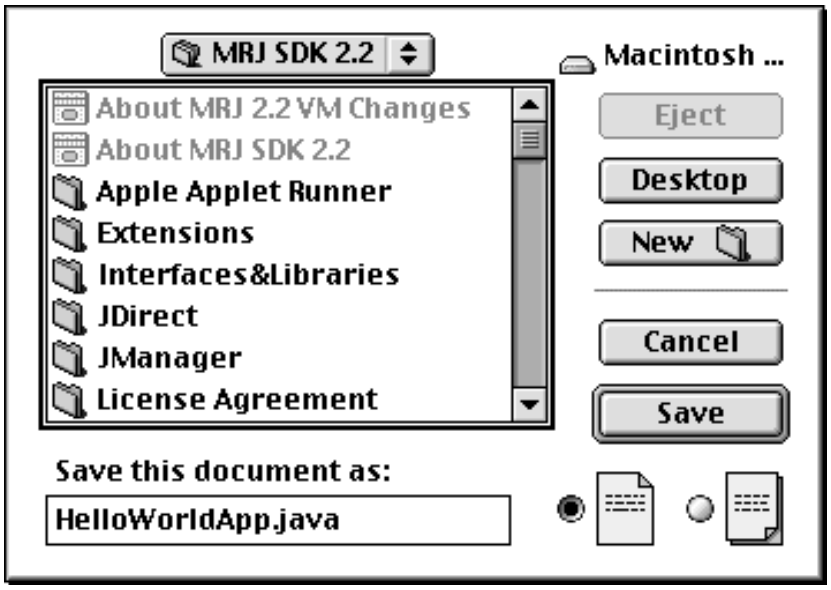

Figure 17 Saving the `HelloWorldApp.java` file.

Now click Save and exit SimpleText.

Compile the Source File

Open the folder `MRJ SDK 2.2` (or whatever you have named your folder) and it should look something like this:.

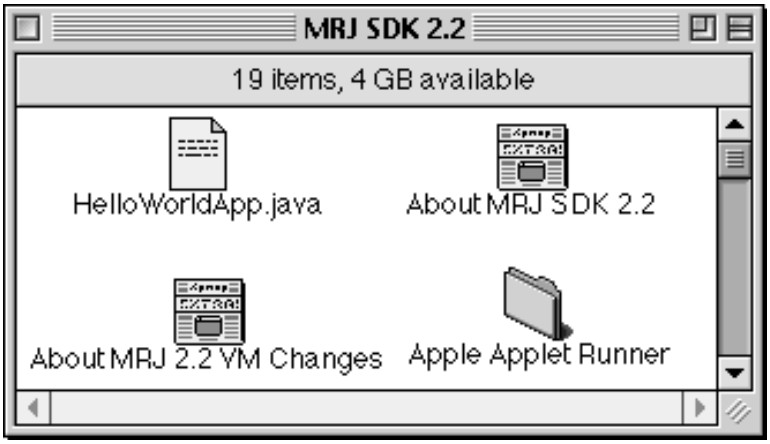

Figure 18 The contents of the `MRJ SDK 2.2` folder.

From the `MRJ SDK 2.2` folder, select `Tools > JDK Tools`. This last folder contains the program `javac`.

Figure 19 The `javac` icon.

Now drag and drop your `HelloWorldApp.java` file onto the `javac` application. `javac` will open and should look like Figure 20.

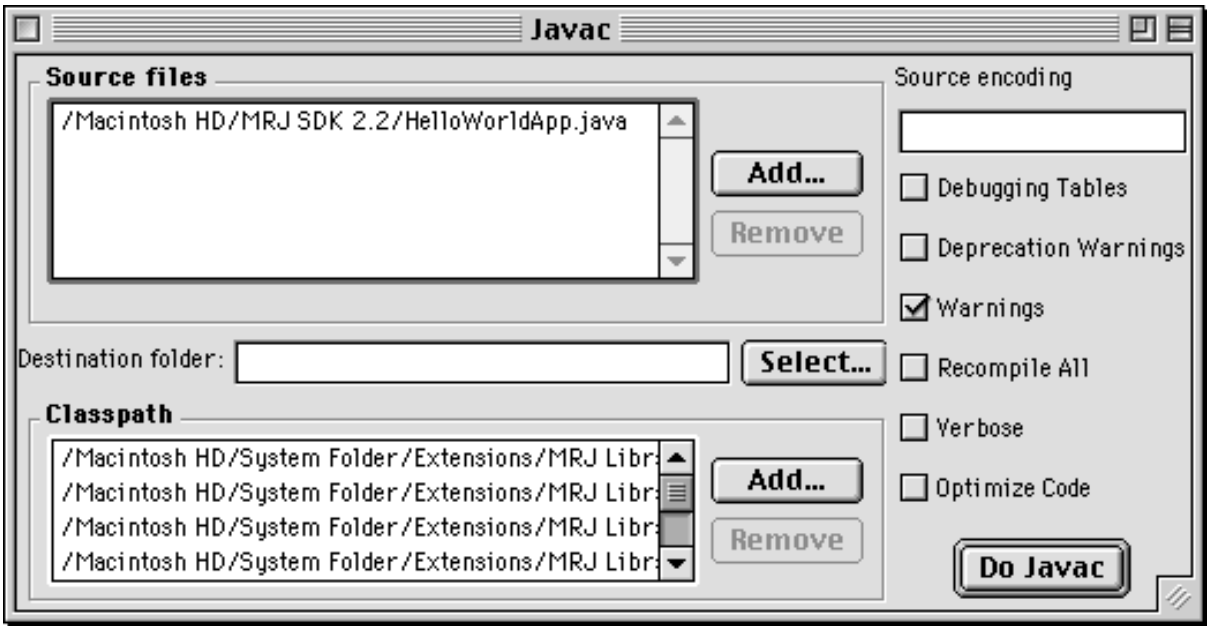

Figure 20 The result of dropping the file `HelloWorldApp.java` onto the `javac` application.

The `Source Files` text area shows the absolute path of the `.java` file we just created. Now there's nothing left to do except press the `Do Javac` button to compile your code. If a message like the one shown in Figure 21 appears without error messages, congratulations. You have successfully compiled your program.

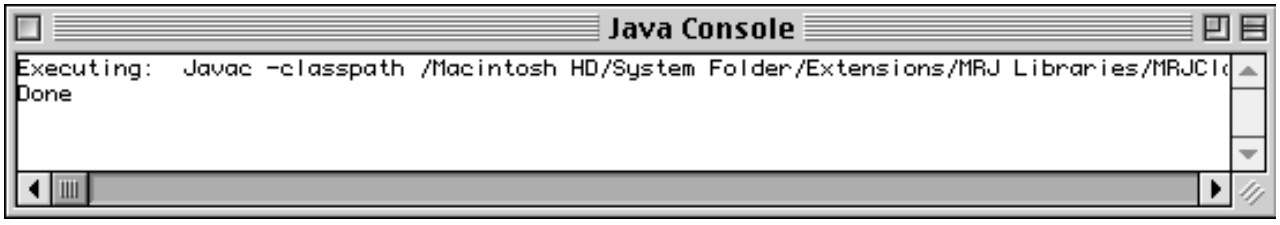

Figure 21 A result of a successful compilation of `HelloWorld.java`.

The compiler has generated a Java bytecode file, `HelloWorldApp.class`. Look in the same folder where you saved the `.java` file to locate the `.class` file. (Figure 22).

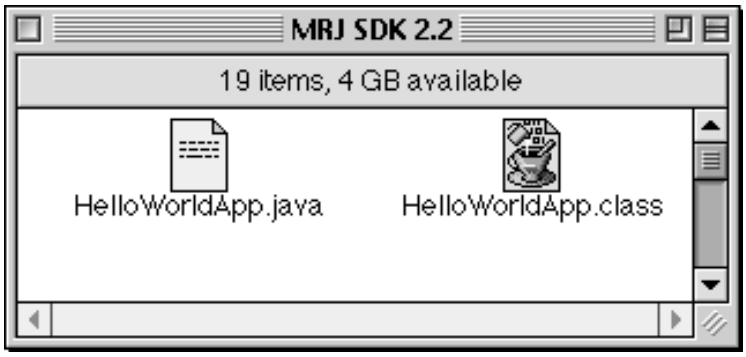

Figure 22 After you compile `HelloWorldApp.java`, the bytecode file, `HelloWorldApp.class`, is created in the same folder.

Now that you have a `.class` file, you can run your program.

Run the Program

From the `MRJ SDK 2.2` folder, select the `Tools > Application Builders > JBindery`. The `JBindery` folder contains the `JBindery` application (Figure 23).

Figure 23 The `JBindery` icon.

Drag and drop the `HelloWorldApp.class` file in the `MRJ SDK 2.2` folder on top of the `JBindery` icon.

Note: A file called `HelloWorld.class` is included with the `JBindery` file. This file is not the one you created.

You should see the dialog shown in Figure 24.

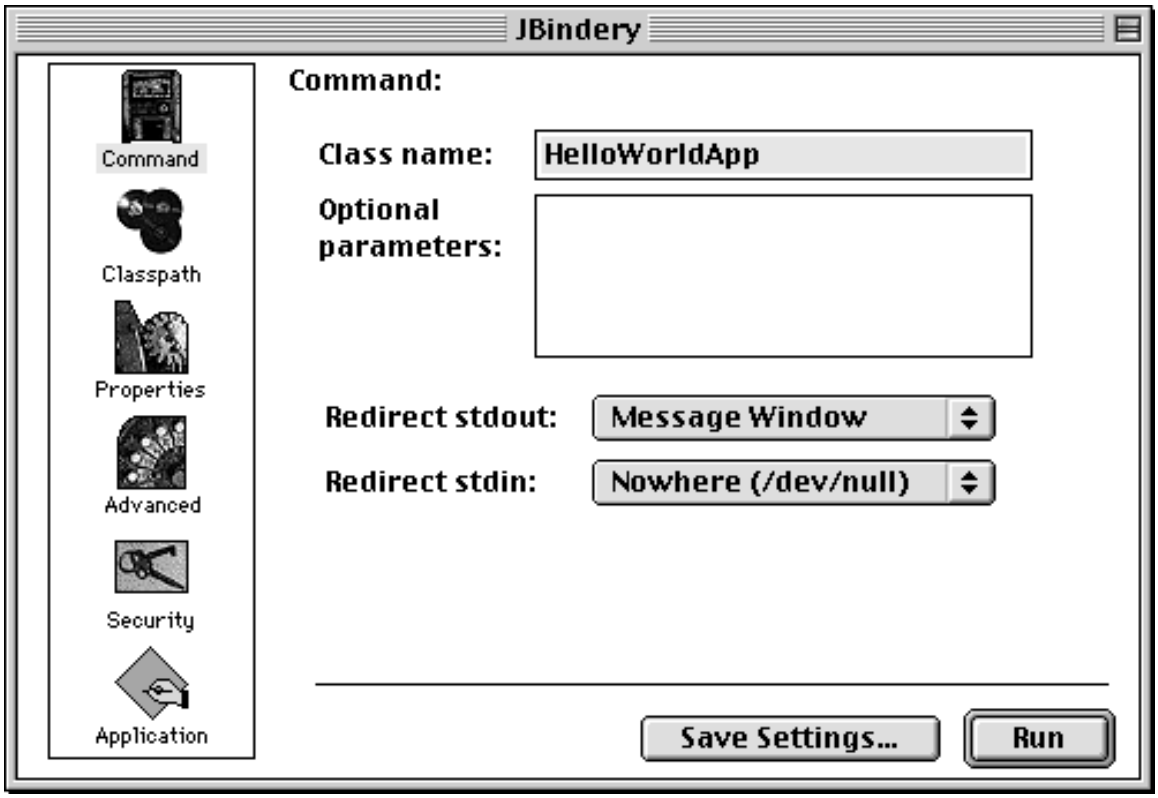

Figure 24 The result of dropping the `HelloWorldApp.class` file onto the `JBindery` program.

Press the `Run` button. Figure 25 shows what you should see.

Figure 25 The result of running the `HelloWorldApp` application.

Congratulations! Your have just run your first program.

Creating Your First Applet

`HelloWorldApp` is an example of an *application*, a standalone program. Now you will create an *applet* called `HelloWorld`, which also displays the greeting “Hello world!” Unlike `Hel-`

loWorldApp, however, the applet runs in a Java-enabled Web browser, such as the HotJava™ browser, Netscape Navigator, or Microsoft Internet Explorer.

To create this applet, you'll perform the basic steps as before: create a source file, compile the source file, and run the program. However, unlike for an application, you must also create an HTML file.

Create a Source File

You have two options to create a source file. You can save the files HelloWorld.java and Hello.html[1] on your computer and avoid a lot of typing. Then you can go straight to the second step of compiling the source file (page 31). Or you can follow these instructions.

First, start SimpleText. In a new document, type the following code:

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

public class HelloWorld extends Applet {
    public void paint(Graphics g) {
        // Display "Hello World!"
        g.drawString("Hello world!", 50, 25);
    }
}
```

Save this code to a file called HelloWorld.java.

Second, you also need an HTML file to accompany your applet. Type the following code into a new SimpleText document:

```
<HTML>
    <HEAD>
        <TITLE>A Simple Program</TITLE>
    </HEAD>
    <BODY>
        Here is the output of my program:
       <APPLET CODE="HelloWorld.class" WIDTH=150 HEIGHT=25>
       </APPLET>
    </BODY>
</HTML>
```

Save this code to a file called Hello.html. Make sure that your files HelloWorld.java and Hello.html are in the same folder.

---

[1] HelloWorld.java and Hello.html are available on this book's CD and online. See Code Samples (page 43).

Compile the Source File

Compile the `HelloWorld.java` source file using `javac` as before. The compiler should generate a bytecode file, `HelloWorld.class`.

Run the Program

Although you can use a Web browser to view your applets, you may find it easier to test your applets by using the simple `Applet Runner` application that comes with the Java platform. To view the `HelloWorld` applet using `Applet Runner`, select `Apple Applet Runner` in the `MRJ SDK 2.2` folder (Figure 25).

Figure 26 The successful execution of the `HelloWorld` applet.

Figure 27 shows what you should see.

Figure 27 The successful execution of the `HelloWorld` applet.

Congratulations! Your applet works. If you encounter errors, see Common Problems and Their Solutions (page 351) in the Appendix to help you fix the problems.

Error Explanation (MacOS)

If you drag and drop your `.java` file on top of the `javac` application and the file is only copied or moved on top of the `javac` application, you need to rebuild your desktop. To rebuild, you must restart you computer and press and hold the `Apple` and `Alt` keys until you a confirmation dialog appears. Answer "Yes" to the question asking if you want to rebuild your desktop. When the rebuilding of your desktop is finished, you should be able to drag and drop the `.java` file onto `javac` to compile your program.

A Closer Look at HelloWorld

Now that you've compiled and run your first program, you may be wondering how it works and how similar it is to other applications or applets. This section will first take a closer look at the `HelloWorldApp` application and then the `HelloWorld` applet. Be aware that the following chapters, Object-Oriented Programming Concepts (page 45) and Language Basics (page 65), will go into much more detail than is presented in this section.

Explanation of an Application

Let's dissect the `HelloWorldApp` application. First, we will look at the comments in the code before we touch on programming concepts such as classes and methods.

```
/**
* The HelloWorldApp class implements an application that
* displays "Hello World!" to the standard output.
*/
public class HelloWorldApp {
    public static void main(String[] args) {
        System.out.println("Hello World!"); //Display the string.
    }
}
```

Comments

The Java programming language supports three kinds of comments:

`/* text */`
: The compiler ignores everything from the opening `/*` to the closing `*/`.

`/** documentation */`
: This style indicates a documentation comment (*doc comment*, for short). As with the first kind of comment, the compiler ignores all the text within the comment. The SDK `javadoc` tool uses doc comments to automatically generate documentation. For more information on `javadoc`, see the tool documentation.[1]

[1] You can find the tool documentation online: http://java.sun.com/j2se/1.3/docs/tooldocs/tools.html

`// text`
The compiler ignores everything from the `//` to the end of the line.

The boldface parts in the following code are comments:

```
/**
 * The HelloWorldApp class implements an application that
 * simply displays "Hello World!" to the standard output.
 */
class HelloWorldApp {
    public static void main(String[] args) {
        System.out.println("Hello World!"); //Display the string.
    }
}
```

Defining a Class

The first boldface line in this listing begins a *class definition block*:

```
/**
 * The HelloWorldApp class implements an application that
 * simply displays "Hello World!" to the standard output.
 */
class HelloWorldApp {
    public static void main(String[] args) {
        System.out.println("Hello World!"); //Display the string.
    }
}
```

A *class* is the basic building block of an object-oriented language, such as the Java programming language. A class is a blueprint that describes the state and the behavior associated with *instances* of that class. When you *instantiate* a class, you create an *object* that has the same states and behaviors as other instances of the same class. The state associated with a class or an object is stored in *member variables*. The behavior associated with a class or an object is implemented with *methods*, which are similar to the functions or procedures in procedural languages, such as C.

A recipe—say, Julia Child's recipe for ratatouille—is like a class. It's a blueprint for making a specific instance of the recipe. Her rendition of ratatouille is one instance of the recipe, and Mary Campione's is (quite) another.

A more traditional example from the world of programming is a class that represents a rectangle. The class defines variables for the origin, width, and height of the rectangle. The class might also define a method that calculates the area of the rectangle. An instance of the rect-

angle class, a rectangle object, contains the information for a specific rectangle, such as the dimensions of the floor of your office or the dimensions of this page.

This is simplest form of a class definition:

```
class Ratatouille {
    . . .                  //class definition block
}
```

The keyword `class` begins the class definition for a class named `Ratatouille`. The variables and the methods of the class are enclosed by the braces that begin and end the class definition block. The `HelloWorldApp` class has no variables and has a single method, named `main`.

The main Method

The entry point of every Java application is its `main` method. When you run an application with the Java interpreter, you specify the name of the class that you want to run. The interpreter invokes the `main` method defined in that class. The `main` method controls the flow of the program, allocates whatever resources are needed, and runs any other methods that provide the functionality for the application.

The boldface lines in the following listing begin and end the definition of the `main` method.

```
/**
 * The HelloWorldApp class implements an application that
 * simply displays "Hello World!" to the standard output.
 */
class HelloWorldApp {
    public static void main(String[] args) {
        System.out.println("Hello World!"); //Display the string.
    }
}
```

Every application must contain a `main` method declared like this:

```
public static void main(String[] args)
```

The `main` method declaration starts with three modifiers:

- `public`: Allows any class to call the `main` method
- `static`: Means the `main` method is associated with the `HelloWorldApp` class as a whole, instead of operating on an instance of the class.
- `void`: Indicates that the `main` method does not return a value

When you invoke the interpreter, you give it the name of the class that you want to run. This class is the application's controlling class and must contain a `main` method. When invoked, the interpreter starts by calling the class's `main` method, which then calls all the other methods required to run the application. If you try to invoke the interpreter on a class that does not have a `main` method, the interpreter can't run your program. Instead, the interpreter displays an error message similar to this:

```
In class NoMain: void main(String argv[]) is not defined
```

As you can see from the declaration of the `main` method, it accepts a single argument: an array of elements of type `String`, like this:

```
public static void main(String[] args)
```

This array is the mechanism through which the Java virtual machine passes information to your application. Each `String` in the array is called a *command-line argument*. Command-line arguments let users affect the operation of the application without recompiling it. The `HelloWorldApp` application ignores its command-line arguments, so there isn't much more to discuss here.

Using Classes and Objects

The `HelloWorldApp` application is about the simplest program you can write that actually does something. Because it is such a simple program, it doesn't need to define any classes except `HelloWorldApp`.

However, the application does *use* another class, `System`, that is part of the Java API. The `System` class provides system-independent access to system-dependent functionality. One feature provided by the `System` class is the *standard output stream*—a place to send text that usually refers to the terminal window in which you invoked the Java interpreter.

Impurity Alert! Using the standard output stream isn't recommended in 100% Pure Java programs. However, it's fine to use during the development cycle. We use it in many of our example programs because otherwise our code would be longer and more difficult to read.

The following boldface line shows `HelloWorldApp`'s use of the standard output stream to display the string `Hello World`:

```
/**
 * The HelloWorldApp class implements an application that
 * simply displays "Hello World!" to the standard output.
 */
```

```
class HelloWorldApp {
    public static void main(String[] args) {
        System.out.println("Hello World!"); //Display the string.
    }
}
```

This one line of code uses both a *class variable* and an *instance method.*

Let's take a look at the first segment of the statement:

```
System.out.println("Hello World!");
```

The construct `System.out` is the full name of the `out` variable in the `System` class. The application never instantiates the `System` class but instead refers to `out` directly through the class. The reason is that `out` is a *class variable*—a variable associated with a class rather than with an object. The Java virtual machine allocates a class variable once per class, no matter how many instances of that class exist. The Java programming language also has the notion of *class methods* used to implement class-specific behaviors.

Although `System`'s `out` variable *is* a class variable, it *refers* to an instance of the `PrintStream` class (another Java API-provided class that implements an easy-to-use output stream). When it is loaded into the application, the `System` class instantiates `PrintStream` and assigns the new `PrintStream` object to the `out` class variable. Now that you have an instance of a class, you can call one of its *instance methods*:

```
System.out.println("Hello World!");
```

An instance method implements behavior specific to a particular object—an instance of a class.

The Java programming language also has *instance variables*. An instance variable is a member variable associated with an object rather than with a class. Each time you instantiate a class, the new object gets its own copy of all the instance variables defined in its class.

If this discussion of member variables, methods, instances, and classes has left you with nothing but questions, the chapters Object-Oriented Programming Concepts (page 45) and Language Basics (page 65) can help.

The Anatomy of an Applet

By following the steps outlined in Creating Your First Applet (page 13 for Win32 and page 20 for UNIX/Linux), you created an applet—a program to be included in HTML pages

and executed in Java-enabled browsers. Remember that an applet is a program that adheres to some conventions that allow it to run within a Java-enabled browser.

Here again is the code for the `HelloWorld` applet:

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

public class HelloWorld extends Applet {
    public void paint(Graphics g) {
        g.drawString("Hello world!", 50, 25);
    }
}
```

Importing Classes and Packages

`HelloWorld.java` begins with two `import` statements which import the `Applet` and `Graphics` classes.

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

public class HelloWorld extends Applet {
    public void paint(Graphics g) {
        g.drawString("Hello world!", 50, 25);
    }
}
```

By importing classes or packages, a class can easily refer to classes in other packages. In the Java programming language, packages are used to group classes, similar to the way libraries group C functions. If you removed the first two lines, you could still compile and run the program, but you could do so only if you changed the rest of the code like this (as shown in boldface):

```
public class HelloWorld extends java.applet.Applet {
    public void paint(java.awt.Graphics g) {
        g.drawString("Hello world!", 50, 25);
    }
}
```

As you can see, importing the `Applet` and `Graphics` classes lets the program refer to them later without any prefixes. The `java.applet.` and `java.awt.` prefixes tell the compiler which packages it should search for the `Applet` and `Graphics` classes. The `java.applet` package contains classes that are essential to applets. The `java.awt` package contains classes used by all programs with a GUI.

You might have noticed that the `HelloWorldApp` application uses the `System` class without any prefix, yet it does not import the `System` class. The reason is that the `System` class is part of the `java.lang` package, and everything in the `java.lang` package is automatically imported into every program written in the Java programming language.

You can import not only individual classes but also entire packages. Here's an example:

```
import java.applet.*;
import java.awt.*;

public class HelloWorld extends Applet {
    public void paint(Graphics g) {
        g.drawString("Hello world!", 50, 25);
    }
}
```

Every class is in a package. If the source code for a class doesn't have a `package` statement at the top declaring in which package the class is, the class is in the *default package*. Almost all the example classes in this tutorial are in the default package.

Within a package, all classes can refer to one another without prefixes. For example, the `java.awt` package's `Component` class refers to the same package's `Graphics` class without any prefixes and without importing the `Graphics` class.

Defining an Applet Subclass

The first boldface line of the following listing begins a block that defines the `HelloWorld` class:

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

public class HelloWorld extends Applet {
    public void paint(Graphics g) {
        g.drawString("Hello world!", 50, 25);
    }
}
```

The `extends` keyword indicates that `HelloWorld` is a subclass of the `Applet` class. In fact, every applet must define a subclass of the `Applet` class. Applets inherit a great deal of functionality from the `Applet` class, ranging from the ability to communicate with the browser to the ability to present a graphical user interface (GUI). You will learn more about subclasses in the chapter Object-Oriented Programming Concepts (page 45).

Implementing Applet Methods

The boldface lines of the following listing implement the `paint` method:

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

public class HelloWorld extends Applet {
    public void paint(Graphics g) {
        g.drawString("Hello world!", 50, 25);
    }
}
```

The `HelloWorld` applet implements just one method: `paint`. Every applet should implement at least one of the following methods: `init`, `start`, or `paint`. Unlike Java applications, applets do *not* need to implement a `main` method.

Applets are designed to be included in HTML pages. Using the `<APPLET>` HTML tag, you specify (at a minimum) the location of the `Applet` subclass and the dimensions of the applet's on-screen display area. The applet's coordinate system starts at (0,0), which is at the upper-left corner of the applet's display area. In the previous code snippet, the string `Hello world!` is drawn starting at location (50,25), which is at the bottom of the applet's display area.

Running an Applet

When a Java-enabled browser encounters an `<APPLET>` tag, it reserves on-screen space for the applet, loads the `Applet` subclass onto the computer on which it is executing, and creates an instance of the `Applet` subclass.[1]

The bold lines of the following listing comprise the `<APPLET>` tag that includes the `HelloWorld` applet in an HTML page:

```
<HTML>
    <HEAD>
        <TITLE> A Simple Program </TITLE>
    </HEAD>
    <BODY>
         Here is the output of my program:
```

[1] You might use other tags to include applets, such as `<OBJECT>` or `<EMBED>`, but we show the `<APPLET>` tag for simplicity.

```
        <APPLET CODE="HelloWorld.class" WIDTH=150 HEIGHT=25>
        </APPLET>
    </BODY>
</HTML>
```

The <APPLET> tag specifies that the browser should load the class whose compiled code (bytecodes) is in the file named `HelloWorld.class`. The browser looks for this file in the same directory as the HTML document that contains the tag.

When the browser finds the class file, it loads it over the network, if necessary, onto the computer on which the browser is running. The browser then creates an instance of the class. If you include an applet twice in one HTML page, the browser loads the class file once and creates two instances of the class.

The `WIDTH` and `HEIGHT` attributes are like the attributes of the same name in an <IMG> tag: They specify the size in pixels of the applet's display area. Most browsers do not let the applet resize itself to be larger or smaller than this display area. For example, all of the drawing that the "Hello World" applet does in its `paint` method occurs within the 150 x 25-pixel display area reserved by the <APPLET> tag.

Questions and Exercises

Questions

1. When you compile a program written in the Java language, the compiler converts the human-readable source file into platform-independent code that a Java virtual machine can understand. What is this platform-independent code called?

2. Which of the following is NOT a valid comment:

   a. `/** comment */`

   b. `/* comment */`

   c. `/* comment`

   d. `// comment`

3. What is the URL of the page in The Java Tutorial that describes Khwarazm? (*Hint:* You can find the answer by going to `http://java.sun.com/docs/books/tutorial/` and click on the link to the Search page where you can perform a search.)

4. a. What is the highest version number of the Java 2 SDK, Standard Edition, that is available for download (early-access releases included)? *(Hint:* You can find the answer at `http://java.sun.com/j2se/`)

   b. What is the highest version number for an SDK that you can download and use in shipping products (that is, not an early-access release)?

   c. What is the lowest version number for an SDK that you can download? (Note that "SDK" used to be called "JDK.")

5. a. Which bug has the highest number of votes at the Java Developer Connection? Give the bug number, description, and number of votes. (*Hint:* Look for the answer at `http://developer.java.sun.com/developer/bugParade/`)

   b. Does the bug report give a workaround? If so, what is it?

6. What is the first thing you should check if the interpreter returns the error: `Exception in thread "main" java.lang.NoClassDefFoundError:HelloWorldApp.java.`

Exercises

1. Change the `HelloWorldApp.java` program so that it displays `Hola Mundo!` instead of `Hello World!`

2. Get the following file from the online Tutorial:

   `http://java.sun.com/docs/books/tutorial/getStarted/QandE-1dot1/Useless.java`

   Compile and run this program. What is the output?

3. You can find a slightly modified version of `HelloWorldApp` here:

   `http://java.sun.com/docs/books/tutorial/getStarted/QandE-1dot1/HelloWorldApp2.java`

   The program has an error. Fix the error so that the program successfully compiles and runs. What was the error?

4. Change the height of the `HelloWorld` applet from `25` to `50`. Describe what the modified applet looks like.

5. Download the following two source files:

   `http://java.sun.com/docs/books/tutorial/getStarted/QandE-1dot1/FirstClass.java`

   `http://java.sun.com/docs/books/tutorial/getStarted/QandE-1dot1/SecondClass.java`

   Compile the files, and then run the resulting program. What is the output? If you have trouble compiling the files, but have successfully compiled before, try unsetting the `CLASSPATH` environment variable as described in Classpath Help (page 442) and then compile again.

Answers

You can find answers to these Questions and Exercises online:

`http://java.sun.com/docs/books/tutorial/getStarted/QandE-1dot1/answers.html`

Code Samples

Table 1 lists the code samples used in this chapter and where you can find the code online and on the CD that accompanies this book.

Table 1 Code Samples in Getting Started

Code Sample (where discussed)	**CD Location**	**Online Location**
`HelloWorldApp.java` (page 9), (page 18), and (page 25)	`/tutorial/getStarted/application/example/HelloWorldApp.java`	`http://java.sun.com/docs/books/tutorial/getStarted/application/example/HelloWorldApp.java`
`HelloWorld.java` (page 13), (page 21), and (page 30)	`/tutorial/getStarted/applet/example/HelloWorld.java`	`http://java.sun.com/docs/books/tutorial/getStarted/applet/example/HelloWorld.java`
`HelloWorld.html` (page 13), (page 21), and (page 30)	`/tutorial/getStarted/applet/example/Hello.html`	`http://java.sun.com/docs/books/tutorial/getStarted/applet/example/Hello.html`

Note: The section Common Problems and Their Solutions (page 351) in the Appendix contains solutions to common problems that Tutorial readers have encountered.