# Introducing Java

This chapter will give you an appreciation of what the Java language is all about. Understanding the details of what we'll discuss in this chapter is not important at this stage; you will see all of them again in greater depth in later chapters of the book. The intent of this chapter is to introduce you to the general ideas that underpin what we'll be covering through the rest of the book, as well as the major contexts in which Java programs can be used and the kind of program that is applicable in each context.

In this chapter you will learn:

- The basic characteristics of the Java language.
- How Java programs work on your computer.
- Why Java programs are portable between different computers.
- The basic ideas behind object-oriented programming.
- How a simple Java program looks and how you can run it using the Java Development Kit.
- What HTML is and how it is used to include a Java program in a web page.

## What is Java All About?

Java is an innovative programming language that has become the language of choice for programs that need to run on a variety of different computer systems. First of all Java enables you to write small programs called **applets.** These are programs that you can embed in Internet web pages to provide some intelligence. Being able to embed executable code in a web page introduces a vast range of exciting possibilities. Instead of being a passive presentation of text and graphics, a web page can be interactive in any way that you want. You can include animations, games, interactive transaction processing – the possibilities are almost unlimited.

Of course, embedding program code in a web page creates special security requirements. As an Internet user accessing a page with embedded Java code, you need to be confident that it will not do anything that might interfere with the operation of your computer, or damage the data you have on your system. This implies that execution of the embedded code must be controlled in such a way that it will prevent accidental damage to your computer environment, as well as ensure that any Java code that was created with malicious intent is effectively inhibited. Java implicitly incorporates measures to minimize the possibility of such occurrences arising with a Java applet.

Java also allows you to write large-scale application programs that you can run unchanged on any computer with an operating system environment in which the language is supported. This applies to the majority of computers in use today. You can even write programs that will work both as ordinary applications and as applets.

Java has matured immensely in recent years, particularly with the introduction of Java 2. The breadth of function provided by the standard core Java has grown incredibly. Java provides you with comprehensive facilities for building application with an interactive GUI, extensive image processing and graphics programming facilities, as well as support for accessing relational databases and communicating with remote computers over a network. Release 1.4 of Java added a very important additional capability, the ability to read and write XML. Just about any kind of application can now be programmed effectively in Java, with the implicit plus of complete portability.

## Features of the Java Language

The most important characteristic of Java is that it was designed from the outset to be machine independent. Java programs can run unchanged on any operating system that supports Java. Of course there is still the slim possibility of the odd glitch as you are ultimately dependent on the implementation of Java on any particular machine, but Java programs are intrinsically more portable than programs written in other languages. An application written in Java will only require a single set of sourcecode, regardless of the number of different computer platforms on which it is run. In any other programming language, the application will frequently require the sourcecode to be tailored to accommodate different computer environments, particularly if there is an extensive graphical user interface involved. Java offers substantial savings in time and resources in developing, supporting, and maintaining major applications on several different hardware platforms and operating systems.

Possibly the next most important characteristic of Java is that it is **object oriented**. The object-oriented approach to programming is also an implicit feature of all Java programs, so we will be looking at what this implies later in this chapter. Object-oriented programs are easier to understand, and less time-consuming to maintain and extend than programs that have been written without the benefit of using objects.

Not only is Java object oriented, but it also manages to avoid many of the difficulties and complications that are inherent in some other object-oriented languages, making it easy to learn and very straightforward to use. It lacks the traps and 'gotchas' that arise in some other programming languages. This makes the learning cycle shorter and you need less real-world coding experience to gain competence and confidence. It also makes Java code easier to test.

Java has a built-in ability to support national character sets. You can write Java programs as easily for Greece or Japan, as you can for English speaking countries always assuming you are familiar with the national languages involved, of course. You can even build programs from the outset to support several different national languages with automatic adaptation to the environment in which the code executes.

# Learning Java

Java is not difficult, but there is a great deal to it. The language itself is fairly compact, but very powerful. To be able to program effectively in Java, however, you also need to understand the libraries that go with the language, and these are very extensive. In this book, the sequence in which you learn how the language works, and how you apply it, has been carefully structured so that you can gain expertise and confidence with programming in Java through a relatively easy and painless process. As far as possible, each chapter avoids the use of things you haven't learned about already. A consequence, though, is that you won't be writing Java applications with a graphical user interface right away. While it may be an appealing idea, this would be a bit like learning to swim by jumping in the pool at the deep end. Generally speaking, there is good evidence that by starting in the shallow end of the pool and learning how to float before you try to swim, the chance of drowning is minimized, and there is a high expectation that you will end up a competent swimmer.

## Java Programs

As we have already noted, there are two kinds of programs you can write in Java. Programs that are to be embedded in a web page are called Java **applets**, and normal standalone programs are called Java **applications**. You can further subdivide Java applications into **console applications,** which only support character output to your computer screen (to the command line on a PC under Windows, for example), and **windowed Java applications** that can create and manage multiple windows. The latter use the typical graphical user interface (GUI) mechanisms of window-based programs – menus, toolbars, dialogs and so on.

While we are learning the Java language basics, we will be using console applications as examples to illustrate how things work. These are application that use simple command line input and output. With this approach we can concentrate on understanding the specifics of the language, without worrying about any of the complexity involved in creating and managing windows. Once we are comfortable with using all the features of the Java language, we'll move on to windowed applications and applet examples.

## Learning Java – the Road Ahead

Before starting out, it is always helpful to have an idea of where you are heading and what route you should take, so let's take a look at a brief road map of where you will be going with Java. There are five broad stages you will progress through in learning Java using this book:

1. The first stage is this chapter. It sets out some fundamental ideas about the structure of Java programs and how they work. This includes such things as what object-oriented programming is all about, and how an executable program is created from a Java source file. Getting these concepts straight at the outset will make learning to write Java programs that much easier for you.

2. Next you will learn how statements are put together, what facilities you have for storing basic data in a program, how you perform calculations and how you make decisions based on the results of them. These are the nuts and bolts you need for the next stages.

3. In the third stage you will learn about **classes** – how you define them and how you can use them. This is where you learn the object-oriented characteristics of the language. By the time you are through this stage you will have learned all the basics of how the Java language works so you will be ready to progress further into how you can use it.

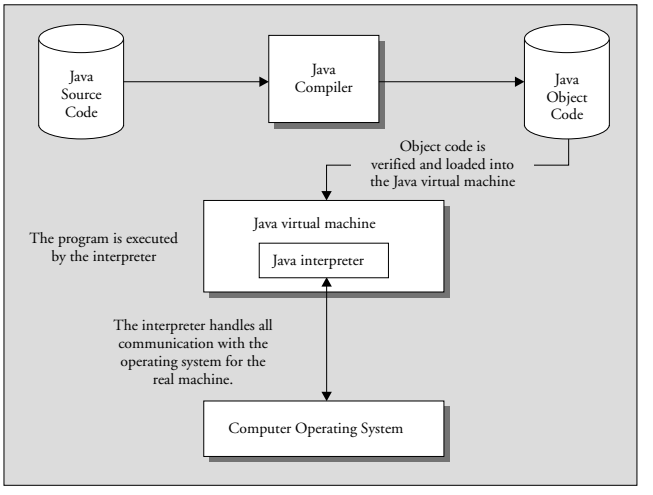
4. In the fourth stage, you will learn how you can segment the activities that your programs carry out into separate tasks that can execute concurrently. This is particularly important for when you want to include several applets in a web page, and you don't want one applet to have to wait for another to finish executing before it can start. You may want a fancy animation to continue running while you play a game, for example, with both programs sitting in the same web page.

5. In the fifth stage you will learn in detail how you implement an application or an applet with a graphical user interface, and how you handle interactions with the user in this context. This amounts to applying the capabilities provided by the Java class libraries. When you finish this stage you will be equipped to write your own fully-fledged applications and applets in Java. At the end of the book, you should be a knowledgeable Java programmer. The rest is down to experience.

Throughout this book we will be using complete examples to explore how Java works. You should create and run all of the examples, even the simplest, preferably by typing them in yourself. Don't be afraid to experiment with them. If there is anything you are not quite clear on, try changing an example around to see what happens, or better still – write an example of your own. If you are uncertain how some aspect of Java that you have already covered works, don't look it up right away – try it out. Making mistakes is a great way to learn.

# The Java Environment

You can run Java programs on a wide variety of computers using a range of operating systems. Your Java programs will run just as well on a PC running Windows 95/98/NT/2000/XP as it will on Linux or a Sun Solaris workstation. This is possible because a Java program does not execute directly on your computer. It runs on a standardized hypothetical computer that is called the **Java virtual machine** or **JVM**, which is emulated inside your computer by a program.



A **Java compiler** converts the Java sourcecode that you write into a binary program consisting of **byte codes**. Byte codes are machine instructions for the Java virtual machine. When you execute a Java program, a program called the **Java interpreter** inspects and deciphers the byte codes for it, checks it out to ensure that it has not been tampered with and is safe to execute, and then executes the actions that the byte codes specify within the Java virtual machine. A Java interpreter can run standalone, or it can be part of a web browser such as Netscape Navigator or Microsoft Internet Explorer where it can be invoked automatically to run applets in a web page.

Because your Java program consists of byte codes rather than native machine instructions, it is completely insulated from the particular hardware on which it is run. Any computer that has the Java environment implemented will handle your program as well as any other, and because the Java interpreter sits between your program and the physical machine, it can prevent unauthorized actions in the program from being executed.

In the past there has been a penalty for all this flexibility and protection in the speed of execution of your Java programs. An interpreted Java program would typically run at only one tenth of the speed of an equivalent program using native machine instructions. With present Java machine implementations, much of the performance penalty has been eliminated, and in programs that are not computation intensive – which is usually the case with the sort of program you would want to include in a web page, for example – you really wouldn't notice this anyway. With the JVM that is supplied with the current Java 2 System Development Kit (SDK) available from the Sun web site, there are very few circumstances where you will notice any appreciable degradation in performance compared to a program compiled to native machine code.

## Java Program Development

There are a number of excellent professional Java program development environments available, including products from Sun, Borland and Symantec. These all provide very friendly environments for creating and editing your sourcecode, and compiling and debugging your programs. These are powerful tools for the experienced programmer, but for learning Java using this book, I recommend that you resist the temptation to use any of these, especially if you are relatively new to programming. Instead, stick to using the Java 2 SDK from Sun together with a suitable simple editor for creating your sourcecode. The professional development systems tend to hide a lot of things you need to understand, and also introduce complexity that you really are better off without while you are learning. These products are intended primarily for knowledgeable and experienced programmers, so start with one when you get to the end of the book.

You can download the SDK from Sun for a variety of hardware platforms and operating systems, either directly from the Sun Java web site at http://java.sun.com (for Windows, Solaris, and Linux operating systems), or from sites that you can link to from there. The SDK we are going to use is available from http://java.sun.com/j2se/1.4. For instance a version of the SDK for Mac OS is available from http://devworld.apple.com/java/.

There is one aspect of terminology that sometimes causes confusion – the SDK used to be known as the JDK – the Java Development kit. If you see JDK this generally means the same as SDK. When you install the Java 2 SDK, you will see the old terminology survives in the name of the root directory where the SDK is installed, currently `/jdk1.4`.

I would urge you to install the SDK even if you do use one or other of the interactive development environments that are available. The SDK provides an excellent reference environment that you can use to check out problems that may arise. Not only that, your programs will only consist of the code that you write plus the classes from the Java libraries that you use. Virtually all commercial Java development systems provide pre-built facilities of their own to speed development. While this is very helpful for production program development, it really does get in the way when you are trying to learn Java.

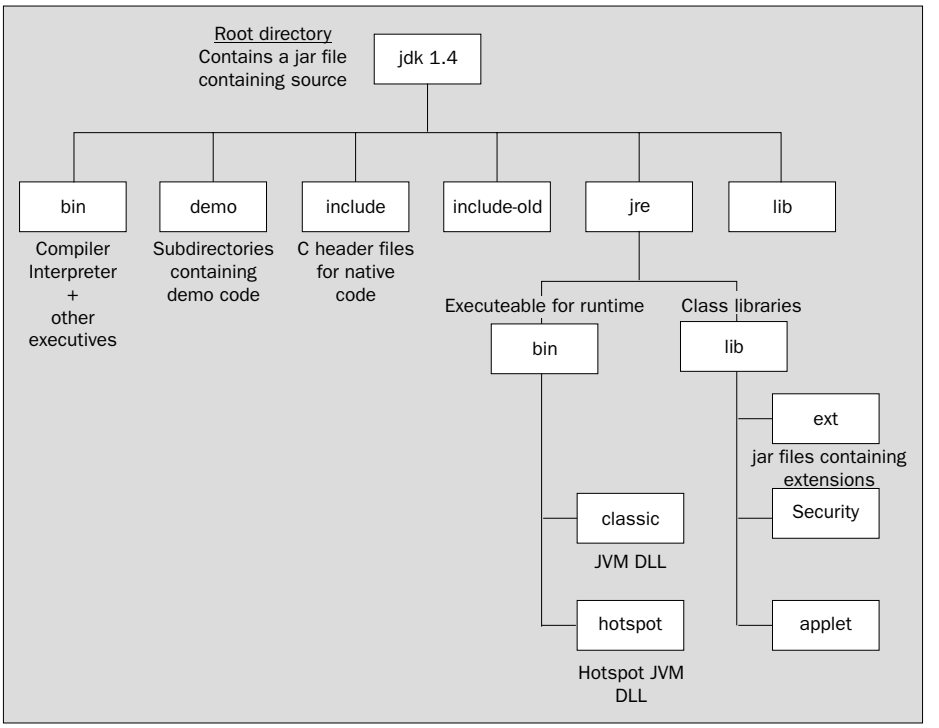
A further consideration is that the version of Java supported by a commercial Java product is not always the most recent. This means that some features of the latest version of Java just won't work. If you really do prefer to work with a commercial Java development system for whatever reason, and you have problems with running a particular example from the book, try it out with the SDK. The chances are it will work OK.

To make use of the SDK you will need a plain text editor. Any editor will do as long as it does not introduce formatting codes into the contents of a file. There are quite a number of shareware and freeware editors around that are suitable, some of which are specific to Java, and you should have no trouble locating one. I find the JCreator editor is particularly good. There's a free version and a fee version with more functionality but the free version is perfectly adequate for learning. You can download a free copy from `http://www.jcreator.com`. A good place to start looking if you want to explore what is available is the `http://www.download.com` web site.

## *Installing the SDK*

You can obtain detailed instructions on how to install the SDK for your particular operating system from the Sun web site, so I won't go into all the variations for different systems here. However, there are a few things to watch out for that may not leap out from the pages of the installation documentation.

First of all, the SDK and the documentation are separate and you install them separately. The SDK for Windows is distributed as a `.exe` file that you just execute to start installation. The documentation for the SDK consists of a large number of HTML files structured in a hierarchy that are distributed in a ZIP archive. You will find it easier to install the SDK first, followed by the documentation. If you install the SDK to drive `C:` under Windows, the directory structure shown in the diagram will be created.




The `jdk1.4` directory in the diagram is sometimes referred to as the **root directory** for Java. In some contexts it is also referred to as the **Java home directory**. If you want the documentation installed in the hierarchy shown above, then you should now extract the documentation from the archive to the `jdk1.4` directory. This corresponds to `C:\jdk1.4` if you installed the SDK to your `C:` drive. This will create a new subdirectory, `docs`, to the `jdk1.4` root directory, and install the documentation files in that. To look at the documentation you just open the `index.html` file that is in the `docs` subdirectory.

You don't need to worry about the contents of most of these directories, at least not when you get started, but you should add the path for the `jdk1.4\bin` directory to the paths defined in your `PATH` environment variable. That way you will be able to run the compiler and the interpreter from anywhere without having to specify supplying the path to it. If you installed the SDK to `C:`, then you need to add the path `C:\jdk1.4\bin`. A word of warning – if you have previously installed a commercial Java development product, check that it has not modified your `PATH` environment variable to include the path to its own Java executables.

If it has, when you try to run the Java compiler or interpreter, you are likely to get the versions supplied with the commercial product rather that those that came with the SDK. One way to fix this is to remove the path or paths that cause the problem. If you don't want to remove the paths that were inserted for the commercial product, you will have to use the full path specification when you want to run the compiler or interpreter from the SDK. The `jre` directory contains the Java Runtime facilities that are used when you execute a Java program. The classes in the Java libraries are stored in the `jre\lib` directory. They don't appear individually though. They are all packaged up in the archive, `rt.jar`. Leave this alone. The Java Runtime takes care of retrieving what it needs from the archive when your program executes.

The CLASSPATH environment variable is a frequent source of problems and confusion to newcomers to Java. The current SDK does ***NOT*** require CLASSPATH to be defined, and if it has been defined by some other Java version or system, it is likely to cause problems. Commercial Java development systems and versions of the Java Development Kit prior to 1.2 may well define the CLASSPATH environment variable, so check to see whether CLASSPATH has been defined on your system. If it has and you no longer have whatever defined it installed, you should delete it. If you have to keep the CLASSPATH environment variable – maybe because you want to keep the system that defined it or you share the machine with someone who needs it – you will have to use a command line option to define CLASSPATH temporarily whenever you compile or execute your Java code. We will see how to do this a little later in this chapter.

#### *Extracting the Sourcecode for the Class Libraries*

The sourcecode for the class libraries is included in the archive `src.zip` that you will find in the `jdk1.4` root directory. Browsing this source can be very educational, and it can also be helpful when you are more experienced with Java in giving a better understanding of how things works – or when they don't, why they don't. You can extract the source files from the archive using the `Winzip` utility or any other utility that will unpack `.zip` archives – but be warned – there's a lot of it and it takes a while!

Extracting the contents of `src.zip` to the root directory `\jdk1.4` will create a new subdirectory, `src`, and install the sourcecode in subdirectories to this. To look at the sourcecode, just open the `.java` file that you are interested in, using any plain text editor.

## *Compiling a Java Program*

Java sourcecode is always stored in files with the extension `.java`. Once you have created the sourcecode for a program and saved it in a `.java` file, you need to process the source using a Java compiler. Using the compiler that comes with the JDK, you would make the directory that contains your Java source file the current directory, and then enter the following command:

```
javac -source 1.4 MyProgram.java
```

Here, `javac` is the name of the Java compiler, and `MyProgram.java` is the name of the program source file. This command assumes that the current directory contains your source file. If it doesn't the compiler won't be able to find your source file. The `-source` command line option with the value `1.4` here tells the compiler that you want the code compiled with the SDK 1.4 language facilities. This causes the compiler to support a facility called **assertions**, and we will see what these are later on. If you leave this option out, the compiler will compile the code with SDK 1.3 capabilities so if the code uses assertions, these will be flagged as errors.

If you need to override an existing definition of the **`CLASSPATH`** environment variable – perhaps because it has been set by a Java development system you have installed, the command would be:

```
javac -source 1.4 -classpath . MyProgram.java
```

The value of **`CLASSPATH`** follows the **`-classpath`** specification and is just a period. This defines just the path to the current directory, whatever that happens to be. This means that the compiler will look for your source file or files in the current directory. If you forget to include the period, the compiler will not be able to find your source files in the current directory. If you include the **`-classpath .`** command line option in any event, it will do no harm.

Note that you should avoid storing your source files within the directory structure that was created for the SDK, as this can cause problems. Set up a separate directory of your own to hold the sourcecode for a program and keep the code for each program in its own directory.

Assuming your program contains no errors, the compiler generates a byte code program that is the equivalent of your source code. The compiler stores the byte code program in a file with the same name as the source file, but with the extension `.class`. Java executable modules are always stored in a file with the extension `.class`. By default, the `.class` file will be stored in the same directory as the source file.

The command line options we have introduced here are by no means all the options you have available for the compiler. You will be able to compile all of the examples in the book just knowing about the options we have discussed. There is a comprehensive description of all the options within the documentation for the SDK. You can also specify the **`-help`** command line option to get a summary of the standard options you can use.

If you are using some other product to develop your Java programs, you will probably be using a much more user-friendly, graphical interface for compiling your programs that won't involve entering commands such as that shown above. The file name extensions for your source file and the object file that results from it will be just the same however.

## Executing a Java Application

To execute the byte code program in the `.class` file with the Java interpreter in the SDK, you make the directory containing the .class file current, and enter the command:

```
java -enableassertions MyProgram
```

Note that we use **`MyProgram`** to identify the program, ***NOT*** `MyProgram.class`. It is a common beginner's mistake to use the latter by analogy with the compile operation. If you put a `.class` file extension on `MyProgram`, your program won't execute and you will get an error message:

```
Exception in thread "main" java.lang.NoClassDefFoundError: MyProgram/class
```

While the compiler expects to find the name of your source file, the java interpreter expects the name of a class, which is **`MyProgram`** in this case, not the name of a file. The `MyProgram.class` file contains the **`MyProgram`** class. We will explain what a class is shortly.

The **`-enableassertions`** option is necessary for SDK1.4 programs that use assertions, but since we will be using assertions once we have learned about them it's a good idea to get into the habit of always using this option. You can abbreviate the **`-enableassertions`** option to **`-ea`** if you wish.

If you want to override an existing **`CLASSPATH`** definition, the option is the same as with the compiler. You can also abbreviate **`-classpath`** to **`-cp`** with the Java interpreter, but strangely, this abbreviation does not apply to the compiler. Here's how the command would look:

```
java -ea -cp . MyProgram
```

To execute your program, the Java interpreter analyzes and then executes the byte code instructions. The Java virtual machine is identical in all computer environments supporting Java, so you can be sure your program is completely portable. As we already said, your program will run just as well on a Unix Java implementation as it will on that for Windows 95/98/NT/2000/XP, for Solaris, Linux, OS/2, or any other operating system that supports Java. (Beware of variations in the level of Java supported though. Some environments, such as the Macintosh, tend to lag a little, so implementations for Java 2 will typically be available later than under Windows or Solaris.)

### *Executing an Applet*

Note that the Java compiler in the SDK will compile both applications and applets. However, an applet is not executed in the same way as an application. You must embed an applet in a web page before it can be run. You can then execute it either within a Java 2-enabled web browser, or by using the `appletviewer`, a bare-bones browser provided as part of the SDK. It is a good idea to use the `appletviewer` to run applets while you are learning. This ensures that if your applet doesn't work, it is almost certainly your code that is the problem, rather than some problem in integration with the browser.

If you have compiled an applet and you have included it in a web page stored as `MyApplet.html` in the current directory on your computer, you can execute it by entering the command:

```
appletviewer MyApplet.html
```

So how do you put an applet in a web page?

## The Hypertext Markup Language

The HyperText Markup Language, or **HTML** as it is commonly known, is used to define a web page. If you want a good, compact, reference guide to HTML, I recommend the book *Instant HTML Programmer's Reference* (Wrox Press, ISBN 1-861001-56-8). Here we will gather just enough on HTML so that you can run a Java applet.

When you define a web page as an HTML document, it is stored in a file with the extension `.html`. An HTML document consists of a number of elements, and each element is identified by **tags**. The document will begin with `<html>` and end with `</html>`. These delimiters, `<html>` and `</html>`, are tags, and each element in an HTML document will be enclosed between a similar pair of tags between angle brackets. All element tags are case insensitive, so you can use uppercase or lowercase, or even a mixture of the two, but by convention they are capitalized so they stand out from the text. Here is an example of an HTML document consisting of a title and some other text:

```
<html>
  <head>
    <title>This is the title of the document</title>
  </head>
  <body>
    You can put whatever text you like here. The body of a document can contain
    all kinds of other HTML elements, including <B>Java applets</B>. Note how each
    element always begins with a start tag identifying the element, and ends with
    an end tag that is the same as the start tag but with a slash added. The pair
    of tags around 'Java applets' in the previous sentence will display the text
    as bold.
```

```
    </body>
</html>
```

There are two elements that can appear directly within the <html> element, a <head> element and a <body> element, as in the example above. The <head> element provides information about the document, and is not strictly part of it. The text enclosed by the <title> element tags that appears here within the <head> element, will be displayed as the window title when the page is viewed.

Other element tags can appear within the <body> element, and they include tags for headings, lists, tables, links to other pages and Java applets. There are some elements that do not require an end tag because they are considered to be empty. An example of this kind of element tag is <hr/>, which specifies a horizontal rule, a line across the full width of the page. You can use the <hr/> tag to divide up a page and separate one type of element from another. You will find a comprehensive list of available HTML tags in the book I mentioned earlier.

## Adding an Applet to an HTML Document

For many element tag pairs, you can specify an **element attribute** in the starting tag that defines additional or qualifying data about the element. This is how a Java applet is identified in an <applet> tag. Here is an example of how you might include a Java applet in an HTML document:

```
<html>
  <head>
    <title> A Simple Program </title>
  </head>
  <body>
    <hr/>
    <applet code = "MyFirstApplet.class"  width = 300  height = 200 >
    </applet>
    <hr/>
  </body>
</html>
```

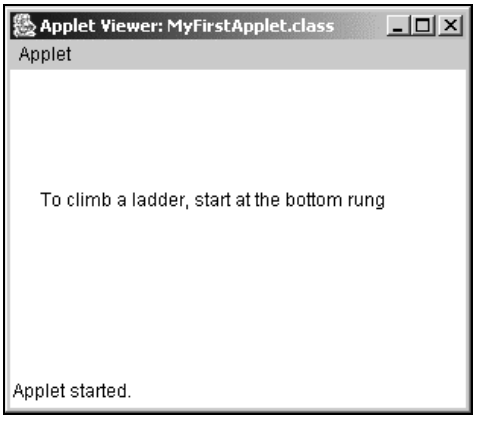
The two shaded lines between tags for horizontal lines specify that the byte codes for the applet are contained in the file MyFirstApplet.class. The name of the file containing the byte codes for the applet is specified as the value for the code attribute in the <applet> tag. The other two attributes, width and height, define the width and height of the region on the screen that will be used by the applet when it executes. These always have to be specified to run an applet. There are lots of other things you can optionally specify, as we will see. Here is the Java sourcecode for a simple applet:

```
import javax.swing.JApplet;
import java.awt.Graphics;

public class MyFirstApplet extends JApplet {

  public void paint(Graphics g) {
    g.drawString("To climb a ladder, start at the bottom rung", 20, 90);
  }
}
```

Note that Java is case sensitive. You can't enter `public` with a capital `P` – if you do the program won't compile. This applet will just display a message when you run it. The mechanics of how the message gets displayed are irrelevant here – the example is just to illustrate how an applet goes into an HTML page. If you compile this code and save the previous HTML page specification in the file `MyFirstApplet.html` in the same directory as the Java applet code, you can run the applet using `appletviewer` from the JDK with the command:

```
appletviewer MyFirstApplet.html
```

This will display a window something like that shown below:




In this particular case, the window is produced under Windows 95/98/NT/2000. Under other operating systems it is likely to look a little different since Java 'takes on' the style of the platform on which it is running. Since the height and width of the window for the applet is specified in pixels, the physical dimensions of the window will depend on the resolution and size of your monitor.

This example won't work with Internet Explorer or Netscape Navigator as neither of these supports Java 2 directly. Let's see what can be done about that.

## *Making Applets Run in Any Browser*

The key to making all your Java applets work with any browser is to make sure the Java 2 Plug-in is installed for each browser that views any of your web pages that contain applets. Making sure that each browser that runs your applet has a Java 2 Plug-in installed is not as hard as you might imagine because you can arrange for it to be automatically downloaded when required – assuming that the computer is online to the Web at the time of course.

The Java Plug-in is a module that can be integrated with Internet Explorer (version 4.0 or later) or Netscape Navigator (version 6.0 or later) to provide full support for Java 2 applets. It supports the use of the `<applet>` tag on any version of Windows from Windows 95 to Windows XP, as well as Linux and Unix.

To enable automatic download of the Java 2 Plug-in with your applets, you need to add some HTML to your web page that invokes a VBScript that handles the download and installation process for the plug-in. To modify our HTML to do this we just need to add one extra tag:

```
<html>
 <head>
  <title> A Simple Program </title>
  <SCRIPT language="VBSCRIPT"
          src="http://java.sun.com/products/plugin/1.4/autodl/autodownload.vbs">
  </SCRIPT>
 </head>
 <body>
  <hr/>
   <applet code = "MyFirstApplet.class"  width = 300  height = 200 >
   </applet>
  <hr/>
 </body>
</html>
```

This makes use of a script that is downloaded from the Sun Java web site. If you want, you can download a copy of the script to your local machine and run it from there. In this case you will need to amend the URL for the `src` attribute in the `<SCRIPT>` to reflect where you have stored the `.vbs` file.

# Object-Oriented Programming in Java

As we said at the beginning of this chapter, Java is an object-oriented language. When you use a programming language that is not object oriented, you must express the solution to every problem essentially in terms of numbers and characters – the basic kinds of data that you can manipulate in the language. In an object-oriented language like Java, things are different. Of course, you still have numbers and characters to work with – these are referred to as the **basic data types** – but you can define other kinds of entities that are relevant to your particular problem. You solve your problem in terms of the entities or objects that occur in the context of the problem. This not only affects how a program is structured, but also the terms in which the solution to your problem is expressed. If your problem concerns baseball players, your Java program is likely to have `BaseballPlayer` objects in it; if you are producing a program dealing with fruit production in California, it may well have objects that are `Oranges` in it. Apart from seeming to be inherently sensible, object-oriented programs are usually easier to understand.

In Java almost everything is an object. If you haven't delved into object-oriented programming before, or maybe because you have, you may feel this is a bit daunting. But fear not. Objects in Java are particularly easy. So easy, in fact, that we are going to start out by understanding some of the ideas behind Java objects right now. In that way you will be on the right track from the outset.

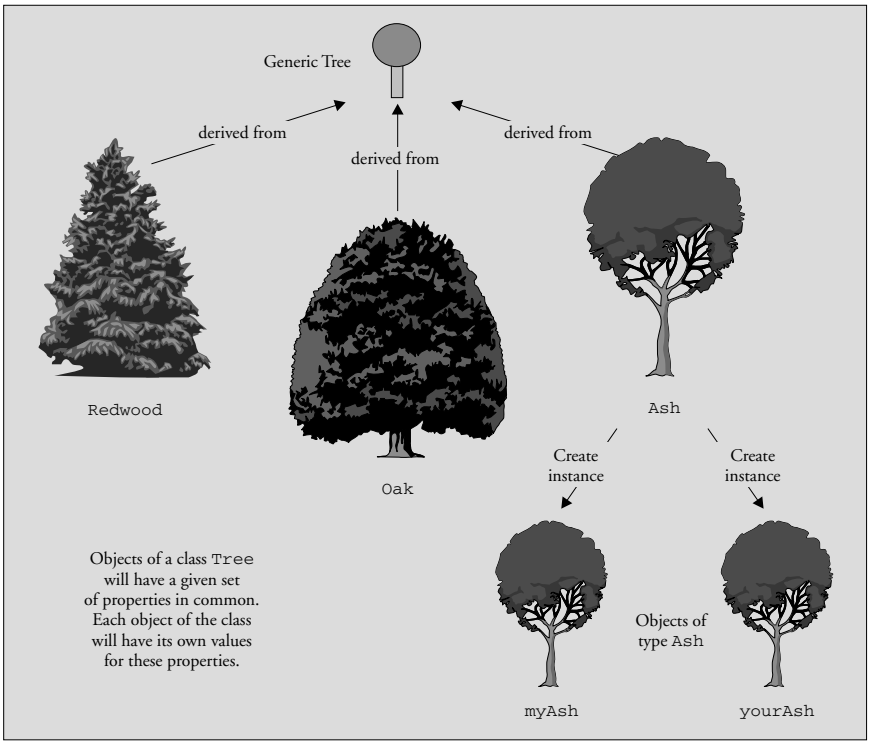
This doesn't mean we are going to jump in with all the precise nitty-gritty of Java that you need for describing and using objects. We are just going to get the concepts straight at this point. We will do this by taking a stroll through the basics using the odd bit of Java code where it helps the ideas along. All the code that we use here will be fully explained in later chapters. Concentrate on understanding the notion of objects first. Then we can ease into the specific practical details as we go along.

## So What Are Objects?

Anything can be thought of as an object. Objects are all around you. You can consider `Tree` to be a particular class of objects: trees in general; although it is a rather abstract class as you would be hard pushed to find an actual occurrence of a totally generic tree. Hence the Oak tree in my yard which I call `myOak`, the Ash tree in your yard which you call `thatDarnedTree`, and a `generalSherman,` the well-known redwood, are actual instances of specific types of tree, subclasses of `Tree` that in this case happen to be `Oak`, `Ash`, and `Redwood`. Note how we drop into the jargon here – **class** is a term that describes a specification for a collection of objects with common properties.

A class is a specification, or template – expressed as a piece of program code – which defines what goes to make up a particular sort of object. A subclass is a class that inherits all the properties of the parent class, but that also includes extra specialization. Of course, you will define a class specification to fit what you want to do. There are no absolutes here. For my trivial problem, the specification of a `Tree` class might just consist of its species and its height. If you are an arboriculturalist, then your problem with trees may require a much more complex class, or more likely a set of classes, that involve a mass of arboreal characteristics.

Every object that your program will use will have a corresponding class definition somewhere for objects of that type. This is true in Java as well as in other object-oriented languages. The basic idea of a class in programming parallels that of classifying things in the real world. It is a convenient and well-defined way to group things together.



An **instance** of a class is a technical term for an existing object of that class. `Ash` is a specification for a type of object and `yourAsh` is an object constructed to that specification, so `yourAsh` would be an instance of the class `Ash`. Once you have a class defined, then you can come up with objects, or instances of that class. This raises the question of what differentiates an object of a given class, an `Ash` class object say, from a `Redwood` object. In other words, what sort of information defines a class?

## What Defines a Class of Objects?

You may have already guessed the answer. A class definition lists all the parameters that you need to define an object of that particular class, at least, so far as your needs go. Someone else might choose a larger or smaller set of parameters to define the same sort of object – it all depends on what you want to do with the class. You will decide what aspects of the objects you need to include to define that particular class of object, and you will choose them depending on the kinds of problems that you want to address using the objects of the class. Let's think about a specific class of objects.

For a class `Hat` for example, you might use just two parameters in the definition. You could include the type of hat as a string of characters such as `"Fedora"` or `"Baseball cap"`, and its size as a numeric value. These parameters that define an object of a class are referred to as **instance variables** or **attributes** of a class, or class **fields**. The instance variables can be basic types of data such as numbers, but they could also be other class objects. For example, the name of a `Hat` object could be of type `String` – the class `String` defines objects that are strings of characters.

Of course there are lots of other things you could include to define a `Hat` if you wanted to, `color` for instance, which might be another string of characters such as `"Blue"`. To specify a class you just decide what set of attributes suit your needs, and those are what you use. This is called **data abstraction** in the parlance of the object-oriented aficionado, because you just abstract the attributes you want to use from the myriad possibilities for a typical object.

In Java the definition of the class `Hat` would look something like:

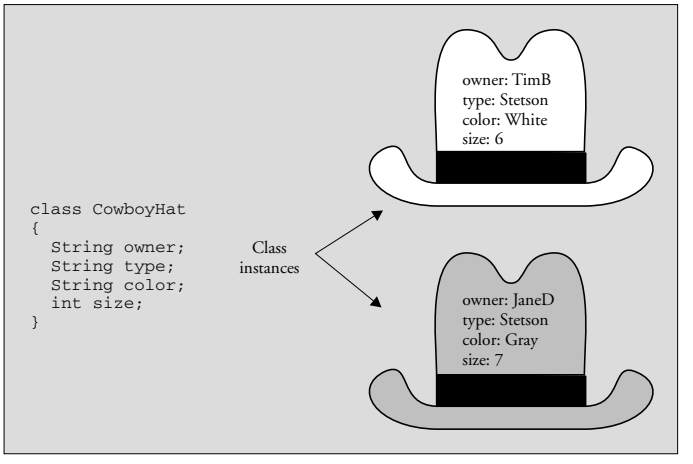
```
class Hat {
  // Stuff defining the class in detail goes here.
  // This could specify the name of the hat, the size,
  // maybe the color, and whatever else you felt was necessary.
}
```

The name of the class follows the word `class`, and the details of the definition appear between the curly braces.

> **Because the word `class` has this special role in Java it is called a** keyword**, and it is reserved for use only in this context. There are lots of other keywords in Java that you will pick up as we go along. You just need to remember that you must not use any of them for any other purposes.**

We won't go into the detail of how the class `Hat` is defined, since we don't need it at this point. The lines appearing between the braces above are not code; they are actually **program comments**, since they begin with two successive forwarded slashes. The compiler will ignore anything on a line that follows two successive forward slashes in your Java programs, so you will use this to add explanations to your programs. Generally the more useful comments you can add to your programs, the better. We will see in Chapter 2 that there are other ways you can write comments in Java.

Each object of your class will have a particular set of values defined that characterize that particular object. You could have an object of type `CowboyHat`, which might be defined by values such as `"Stetson"` for the name of the hat, `"White"` for the color, and the size as 7.



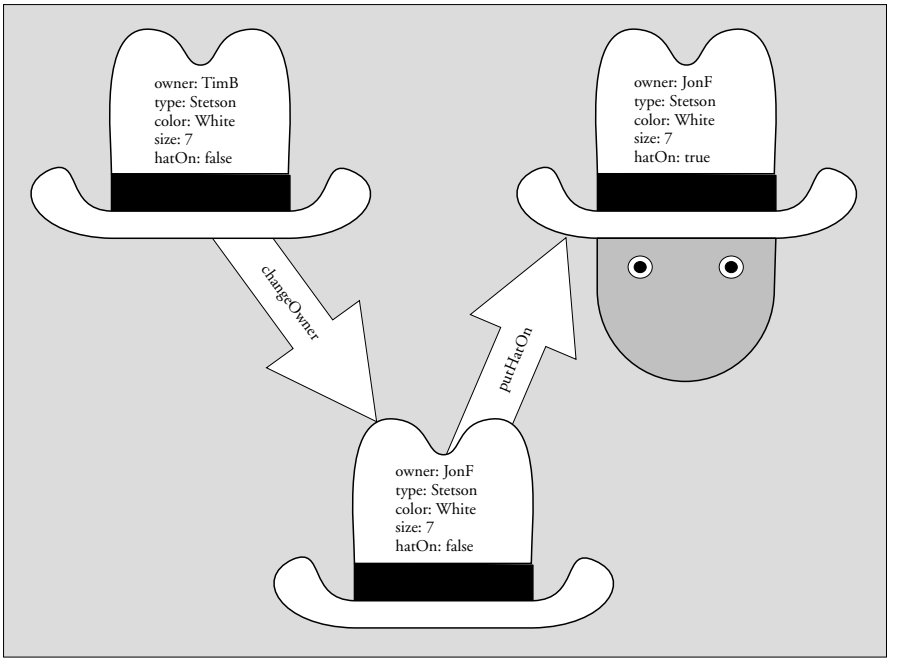

The parameters defining an object are not necessarily fixed values though. You would expect the `name` and `size` attributes for a particular `CowboyHat` object to stay fixed since hats don't usually change their size, but you could have other attributes. You might have `state` for example, which could indicate whether the hat was on or off the owner's head, or even `owner`, which would record the owner's name, so the value stored as the attribute `owner` could be changed when the hat was sold or otherwise transferred to someone else.

### *Operating on Objects*

A class object is not just a collection of various items of data though. The fundamental difference between a class and the complex data types that you find in some other languages is that a class includes more than just data. A class specifies what you can do with an object of the class – that is, it defines the operations that are possible on objects of the class. Clearly for objects to be of any use in a program, you need to decide what you can do with them. This will depend on what sort of objects you are talking about, the attributes they contain, and how you intend to use them.

To take a very simple example, if your objects were numbers, of type `Integer` for example, it would be reasonable to plan for the usual arithmetic operations; add, subtract, multiply and divide, and probably a few others you can come up with. On the other hand it would not make sense to have operations for calculating the area of an `Integer`, boiling an `Integer` or for putting an `Integer` object on. There are lots of classes where these operations would make sense, but not those dealing with integers.

Coming back to our `CowboyHat` class, you might want to have operations that you could refer to as `putHatOn` and `takeHatOff`, which would have meanings that are fairly obvious from their names, and do make sense for `CowboyHat` objects. However, these operations would only be effective if a `CowboyHat` object also had another defining value that recorded whether it was on or off. Then these operations on a particular `CowboyHat` object could set this value for the object. To determine whether your `CowboyHat` was on or off, you would just need to look at this value. Conceivably you might also have an operation `changeOwner` by which you could set the instance variable recording the current owner's name to a new value. The illustration shows two operations applied in succession to a `CowboyHat` object.

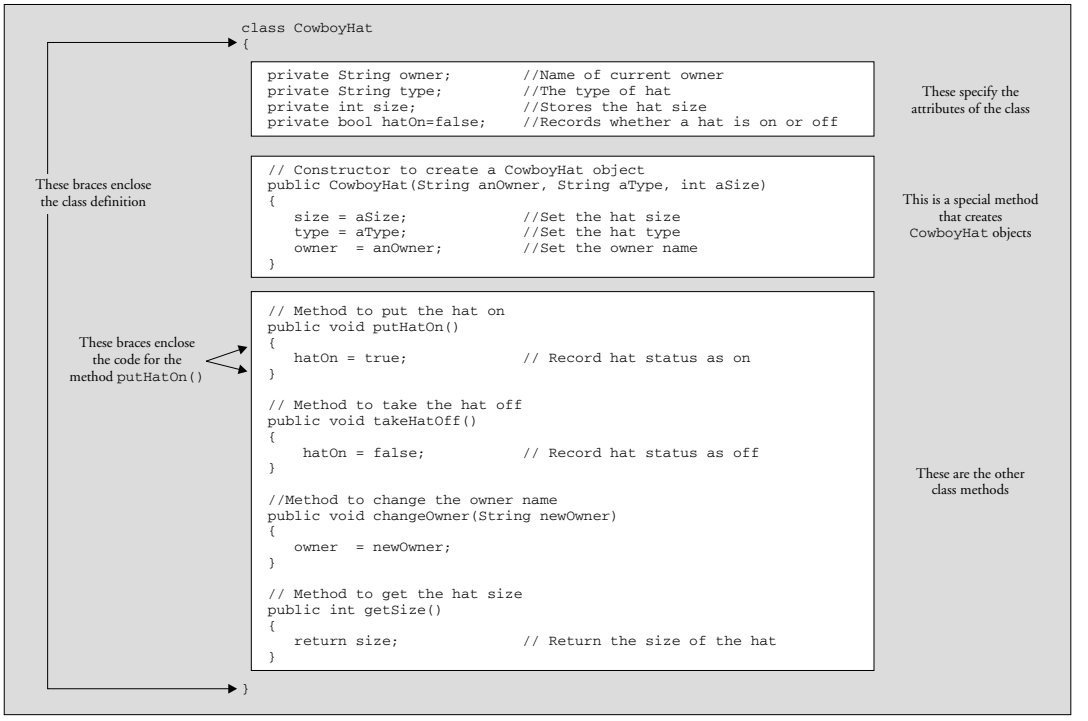



Of course, you can have any operation for each type of object that makes sense for you. If you want to have a `shootHoleIn` operation for `Hat` objects, that's no problem. You just have to define what that operation does to an object.

You are probably wondering at this point how an operation for a class is defined. As we shall see in detail a bit later, it boils down to a self-contained block of program code called a **method** that is identified by the name you give to it. You can pass data items – which can be integers, floating point numbers, character strings or class objects – to a method, and these will be processed by the code in the method. A method may also return a data item as a result. Performing an operation on an object amounts to 'executing' the method that defines that operation for the object.

**Of course, the only operations you can perform on an instance of a particular class are those defined within the class, so the usefulness and flexibility of a class is going to depend on the thought that you give to its definition. We will be looking into these considerations more in Chapter 5.**

Let's take a look at an example of a complete class definition. The code for the class `CowboyHat` we have been talking about might look like the following:




This code would be saved in a file with the name `CowboyHat.java`. The name of a file that contains the definition of a class is always the same as the class name, and the extension will be `.java` to identify that the file contains Java sourcecode.

The code for the class definition appears between the braces following the identification for the class, as shown in the illustration. The code for each of the methods in the class also appears between braces. The class has four instance variables, `owner`, `type`, `size`, and `hatOn`, and this last variable is always initialized as `false`. Each object that is created according to this class specification will have its own independent copy of each of these variables, so each object will have its own unique values for the owner, the hat size, and whether the hat is on or off.

The keyword `private`, which has been applied to each instance variable, ensures that only code within the methods of the class can access or change the values of these directly. Methods of a class can also be specified as `private`. Being able to prevent access to some members of a class from outside is an important facility. It protects the internals of the class from being changed or used incorrectly. Someone using your class in another program can only get access to the bits to which you want them to have access. This means that you can change how the class works internally without affecting other programs that may use it. You can change any of the things inside the class that you have designated as `private`, and you can even change the code inside any of the public methods, as long as the method name and the number and types of values passed to it or returned from it remain the same.

Our `CowboyHat` class also has five methods, so you can do five different things with a `CowboyHat` object. One of these is a special method called a **constructor**, which creates a `CowboyHat` object – this is the method with the name, `CowboyHat`, that is the same as the class name. The items between the parentheses that follow the name of the constructor specify data that is to be passed to the method when it is executed – that is, when a `CowboyHat` object is created.

> **In practice you might need to define a few other methods for the class to be useful; you might want to compare `CowboyHat` objects for example, to see if one was larger than another. However, at the moment you just need to get an idea of how the code looks. The details are of no importance here, as we will return to all this in Chapter 5.**

## Java Program Statements

As you saw in the `CowboyHat` class example, the code for each method in the class appears between braces, and it consists of **program statements**. A semicolon terminates each program statement. A statement in Java can spread over several lines if necessary, since the end of each statement is determined by the semicolon, not by the end of a line. Here is a Java program statement:

```
hatOn = false;
```

If you wanted to, you could also write this as:

```
hatOn =
          false;
```

You can generally include spaces and tabs, and spread your statements over multiple lines to enhance readability if it is a particularly long statement, but sensible constraints apply. You can't put a space in the middle of a name for instance. If you write `hat On`, for example, the compiler will read this as two words.

### Encapsulation

At this point we can introduce another bit of jargon you can use to impress or bore your friends – **encapsulation**. Encapsulation refers to the hiding of items of data and methods within an object. This is achieved by specifying them as `private` in the definition of the class. In the `CowboyHat` class, the instance variables, `owner`, `type`, `size`, and `hatOn` were encapsulated. They were only accessible through the methods defined for the class. Therefore the only way to alter the values they contain is to call a method that does that. Being able to encapsulate members of a class in this way is important for the security and integrity of class objects. You may have a class with data members that can only take on particular values. By hiding the data members and forcing the use of a method to set or change the values, you can ensure that only legal values are set.

We mentioned earlier another major advantage of encapsulation – the ability to hide the implementation of a class. By only allowing limited access to the members of a class, you have the freedom to change the internals of the class without necessitating changes to programs that use the class. As long as the external characteristics of the methods that can be called from outside the class remain unchanged, the internal code can be changed in any way that you, the programmer, want.

A particular object, an instance of `CowboyHat`, will incorporate, or encapsulate, the `owner`, the `size` of the object, and the status of the hat in the instance variable `hatOn`. Only the constructor, and the `putHatOn()`, `takeHatOff()`, `changeOwner()`, and `getSize()` methods can be accessed externally.

> **Whenever we are referring to a method in the text, we will add a pair of parentheses after the method name to distinguish it from other things that have names. Some examples of this appear in the paragraph above. A method always has parentheses in its definition and in its use in a program, as we shall see, so it makes sense to represent it in this way in the text.**

## *Classes and Data Types*

Programming is concerned with specifying how data of various kinds is to be processed, massaged, manipulated or transformed. Since classes define the types of objects that a program will work with, you can consider defining a class to be the same as defining a data type. Thus `Hat` is a type of data, as is `Tree`, and any other class you care to define. Java also contains a library of standard classes that provide you with a whole range of programming tools and facilities. For the most part then, your Java program will process, massage, manipulate or transform class objects.

There are some basic types of data in Java that are not classes, and these are called **primitive types**. We will go into these in detail in the next chapter, but they are essentially data types for numeric values such as 99 or 3.75, for single characters such as 'A' or '?', and for logical values that can be `true` or `false`. Java also has classes that correspond to each of the primitive data types for reasons that we will see later on so there is an `Integer` class that defines objects that encapsulate integers for instance. Every entity in your Java program that is not of a primitive data type will be an object of a class – either a class that you define yourself, a class supplied as part of the Java environment, or a class that you obtain from somewhere else, such as from a specialized support package.

## *Classes and Subclasses*

Many sets of objects that you might define in a class can be subdivided into more specialized subsets that can also be represented by classes, and Java provides you with the ability to define one class as a more specialized version of another. This reflects the nature of reality. There are always lots of ways of dividing a cake – or a forest. `Conifer` for example could be a subclass of the class `Tree`. The `Conifer` class would have all the instance variables and methods of the `Tree` class, plus some additional instance variables and/or methods that make it a `Conifer` in particular. You refer to the `Conifer` class as a **subclass** of the class `Tree`, and the class `Tree` as a **superclass** of the class `Conifer`.

When you define a class such as `Conifer` using another class such as `Tree` as a starting point, the class `Conifer` is said to be **derived** from the class `Tree`, and the class `Conifer` **inherits** all the attributes of the class `Tree`.

### *Advantages of Using Objects*

As we said at the outset, object-oriented programs are written using objects that are specific to the problem being solved. Your pinball machine simulator may well define and use objects of type `Table`, `Ball`, `Flipper`, and `Bumper`. This has tremendous advantages, not only in terms of easing the development process, but also in any future expansion of such a program. Java provides a whole range of standard classes to help you in the development of your program, and you can develop your own generic classes to provide a basis for developing programs that are of particular interest to you.

Because an object includes the methods that can operate on it as well as the data that defines it, programming using objects is much less prone to error. Your object-oriented Java programs should be more robust than the equivalent in a procedural programming language. Object-oriented programs take a little longer to design than programs that do not use objects since you must take care in the design of the classes that you will need, but the time required to write and test the code is sometimes substantially less than that for procedural programs. Object-oriented programs are also much easier to maintain and extend.

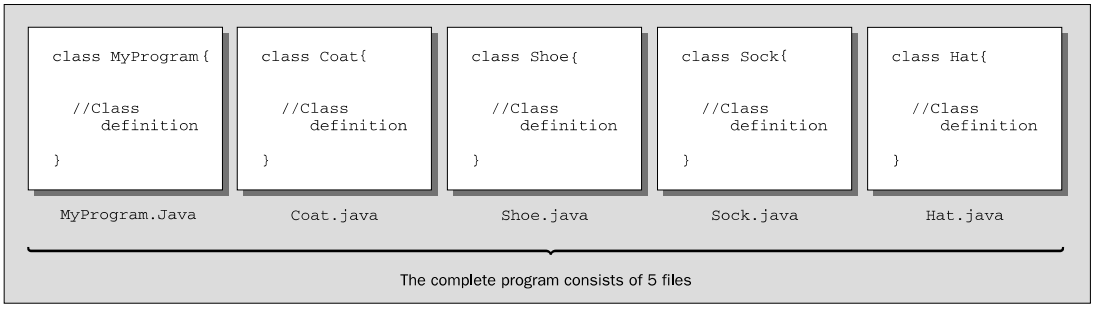
## Java Program Structure

Let's summarize the general nature of how a Java program is structured :

- A Java program always consists of one or more classes.
- You typically put the program code for each class in a separate file, and you must give each file the same name as that of the class that is defined within it.
- A Java source file must also have the extension `.java`.

Thus your file containing the class `Hat` will be called `Hat.java` and your file containing the class `BaseballPlayer` must have the file name `BaseballPlayer.java`.

A typical program will consist of several files as illustrated in the following diagram.




This program clearly majors on apparel with four of the five classes representing clothing. Each source file will contain a class definition, and all of the files that go to make up the program will be stored in the same directory. The source files for your program will contain all the code that you wrote, but this is not everything that is ultimately included in the program. There will also be code from the **Java standard class library**, so let's take a peek at what that can do.

## Java's Class Library

A library in Java is a collection of classes – usually providing related facilities – which you can use in your programs. The Java class library provides you with a whole range of goodies, some of which are essential for your programs to work at all, and some of which make writing your Java programs easier. To say that the standard class library covers a lot of ground would be something of an understatement so we won't be going into it in detail here, but we will be looking into how to apply many of the facilities it provides throughout the book.

Since the class library is a set of classes, it is stored in sets of files where each file contains a class definition. The classes are grouped together into related sets that are called **packages**, and each package is stored in a separate directory. A class in a package can access any of the other classes in the package. A class in another package may or may not be accessible. We will learn more about this in Chapter 5.

The package name is based on the path to the directory in which the classes belonging to the package are stored. Classes in the package `java.lang` for example are stored in the directory path `java\lang` (or `java/lang` under Unix). This path is relative to a particular directory that is automatically known by the Java runtime environment that executes your code. You can also create your own packages that will contain classes of your own that you want to reuse in different contexts, and that are related in some way.

The SDK includes a growing number of standard packages – well over 100 the last time I counted. Some of the packages you will meet most frequently are:

| Package Name | Description |
|---|---|
| `java.lang` | These classes support the basic language features and the handling of arrays and strings. Classes in this package are always available directly in your programs by default because this package is always automatically loaded with your program. |
| `java.io` | Classes for data input and output operations. |
| `java.util` | This package contains utility classes of various kinds, including classes for managing data within collections or groups of data items. |
| `javax.swing` | These classes provide easy-to-use and flexible components for building graphical user interfaces (GUIs). The components in this package are referred to as Swing components. |
| `java.awt` | Classes in this package provide the original GUI components (JDK1.1) as well as some basic support necessary for Swing components. |
| `java.awt.geom` | These classes define 2-dimensional geometric shapes. |
| `java.awt.event` | The classes in this package are used in the implementation of windowed application to handle events in your program. Events are things like moving the mouse, pressing the left mouse button, or clicking on a menu item. |

As noted above, you can use any of the classes from the `java.lang` package in your programs by default. To use classes from the other packages, you will typically use `import` statements to identify the names of the classes that you need from each package. This will allow you to reference the classes by the simple class name. Without an `import` statement you would need to specify the fully qualified name of each class from a package each time you refer to it. As we will see in a moment, the fully qualified name for a class includes the package name as well as the basic class name. Using fully qualified class names would make your program code rather cumbersome, and certainly less readable. It would also make them a lot more tedious to type in.

You can use an `import` statement to import the name of a single class from a package into your program, or all the class names. The two `import` statements at the beginning of the code for the applet you saw earlier in this chapter are examples of importing a single class name. The first was:

```
import javax.swing.JApplet;
```

This statement imports the `JApplet` class name that is defined in the `javax.swing` package. Formally, the name of the `JApplet` class is not really `JApplet` – it is the fully qualified name `javax.swing.JApplet`. You can only use the unqualified name when you import the class or the complete package containing it into your program. You can still reference a class from a package even if you don't import it though – you just need to use the full class name, `javax.swing.JApplet`. You could try this out with the applet you saw earlier if you like. Just delete the two import statements from the file and use the full class names in the program. Then recompile it. It should work the same as before. Thus the fully qualified name for a class is the name of the package in which it is defined, followed by a period, followed by the name given to the class in its definition.

You could import the names of all the classes in the `javax.swing` package with the statement:

```
import javax.swing.*;
```

The asterisk specifies that all the class names are to be imported. Importing just the class names that your sourcecode uses makes compilation more efficient, but when you are using a lot of classes from a package you may find it more convenient to import all the names. This saves typing reams of import statements for one thing. We will do this with examples of Java code in the book to keep the number of lines to a minimum. However, there are risks associated with importing all the names in a package. There may be classes with names that are identical to names you have given to your own classes, which would obviously create some confusion when you compile your code.

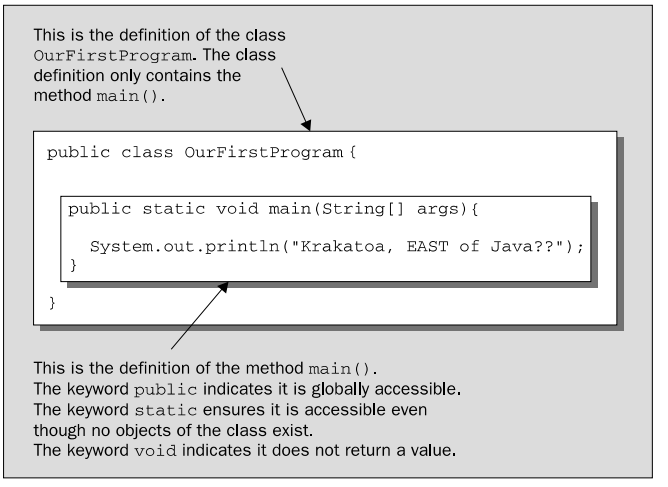
**You will see more on how to use `import` statements in Chapter 5, as well as more about how packages are created and used, and you will be exploring the use of classes from the standard packages in considerable depth throughout the book.**

As we indicated earlier, the standard classes do not appear as files or directories on your hard disk. They are packaged up in a single compressed file, `rt.jar`, that is stored in the `jre/lib` directory. This directory is created when you install the JDK on your computer. A `.jar` file is a **J**ava **ar**chive – a compressed archive of Java classes. The standard classes that your executable program requires are loaded automatically from `rt.jar`, so you don't have to be concerned with it directly at all.

## Java Applications

Every Java application contains a class that defines a method called `main()`. The name of this class is the name that you use as the argument to the Java interpreter when you run the application. You can call the class whatever you want, but the method which is executed first in an application is always called `main()`. When you run your Java application the method `main()` will typically cause methods belonging to other classes to be executed, but the simplest possible Java application program consists of one class containing just the method `main()`. As we shall see below, the `main()` method has a particular fixed form, and if it is not of the required form, it will not be recognized by the Java interpreter as the method where execution starts.

We'll see how this works by taking a look at just such a Java program. You need to enter the program code using your favorite plain text editor, or if you have a Java development system with an editor, you can enter the code for the example using that. When you have entered the code, save the file with the same name as that used for the class and the extension `.java`. For this example the file name will be `OurFirstProgram.java`. The code for the program is:




The program consists of a definition for a class we have called `OurFirstProgram`. The class definition only contains one method, the method `main()`. The first line of the definition for the method `main()` is always of the form:

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

The code for the method appears between the pair of curly braces. Our version of the method has only one executable statement:

```
System.out.println("Krakatoa, EAST of Java??");
```

So what does this statement do? Let's work through it from left to right:

- `System` is the name of a standard class that contains objects that encapsulate the standard I/O devices for your system – the keyboard for command line input and command line output to the display. It is contained in the package `java.lang` so it is always accessible just by using the simple class name, `System`.
- The object `out` represents the standard output stream – the command line on your display screen, and is a data member of the class `System`. The member, `out`, is a special kind of member of the `System` class. Like the method `main()` in our `OurFirstProgram` class, it is `static`. This means that `out` exists even though there are no objects of type `System` (more on this in forthcoming chapters). Using the class name, `System`, separated from the member name `out` by a period – `System.out`, references the `out` member.
- The bit at the rightmost end of the statement, `println("Krakatoa, EAST of Java??")`, calls the `println()` method that belongs to the object `out`, and that outputs the text string that appears between the parentheses to your display. This demonstrates one way in which you can call a class method – by using the object name followed by the method name, with a period separating them. The stuff between the parentheses following the name of a method is information that is passed to the method when it is executed. As we said, for `println()` it is the text we want to output to the command line.

*For completeness, the keywords `public`, `static`, and `void`, that appear in the method definition are explained briefly in the annotations to the program code, but you need not be concerned if these still seem a bit obscure at this point. We will be coming back to them in much more detail later on.*

You can compile this program using the JDK compiler with the command,

```
javac –source 1.4 OurFirstProgram.java
```

Or with the `-classpath` option specified:

```
javac –source 1.4 –classpath . OurFirstProgram.java
```

If it didn't compile, there's something wrong somewhere. Here's a checklist of possible sources of the problem:

- You forgot to include the path to the **`jdk1.4\bin`** directory in your **`PATH`**, or maybe you did not specify the path correctly. This will result in your operating system not being able to find the `javac` compiler that is in that directory.
- You made an error typing in the program code. Remember Java is case sensitive so `OurfirstProgram` is not the same as `OurFirstProgram`, and of course, there must be no spaces in the class name. If the compiler discovers an error it will usually identify the line number in the code where the error was found. In general, watch out for confusing zero, `0`, with a small letter, `o`, or the digit one, `1`, with the small letter `l`. All characters such as periods, commas, and semicolons in the code are essential, and must be in the right place. Parentheses, (), curly braces, {}, and square brackets, [], always come in matching pairs and are not interchangeable.
- The source file name must match the class name exactly. The slightest difference will result in an error. It must have the extension `.java`.

Once you have compiled the program successfully, you can execute it with the command:

```
java -ea OurFirstProgram
```

The `-ea` option is not strictly necessary since this program does not use assertions but if you get used to putting it in, you won't forget it when it is necessary. If you need the `-classpath` option specified:

```
java -ea -classpath . OurFirstProgram
```

Assuming the source file compiled correctly, and the `jdk1.4\bin` directory is defined in your path, the most common reason for the program failing to execute is a typographical error in the class name, `OurFirstProgram`. The second most common reason is writing the file name, `OurFirstProgram.class`, in the command, whereas it should be just the class name, `OurFirstProgram`.

When you run the program, it will display the text:

```
Krakatoa, EAST of Java??
```

# Java and Unicode

Programming to support languages that use anything other than the Latin character set has always been a major problem. There are a variety of 8-bit character sets defined for many national languages, but if you want to combine the Latin character set and Cyrillic in the same context, for example, things can get difficult. If you want to handle Japanese as well, it becomes impossible with an 8-bit character set because with 8 bits you only have 256 different codes so there just aren't enough character codes to go round. Unicode is a standard character set that was developed to allow the characters necessary for almost all languages to be encoded. It uses a 16-bit code to represent a character (so each character occupies two bytes), and with 16 bits up to 65,535 non-zero character codes can be distinguished. With so many character codes available, there is enough to allocate each major national character set its own set of codes, including character sets such as Kanji which is used for Japanese, and which requires thousand of character codes. It doesn't end there though. Unicode supports three encoding forms that allow up to a million additional characters to be represented.

As we shall see in Chapter 2, Java sourcecode is in Unicode characters. Comments, identifiers (names – see Chapter 2), and character and string literals can all use any characters in the Unicode set that represent letters. Java also supports Unicode internally to represent characters and strings, so the framework is there for a comprehensive international language capability in a program. The normal ASCII set that you are probably familiar with corresponds to the first 128 characters of the Unicode set. Apart from being aware that each character occupies two bytes, you can ignore the fact that you are handling Unicode characters in the main, unless of course you are building an application that supports multiple languages from the outset.

# Summary

In this chapter we have looked at the basic characteristics of Java, and how portability between different computers is achieved. We have also introduced the elements of object-oriented programming. There are bound to be some aspects of what we have discussed that you don't feel are completely clear to you. Don't worry about it. Everything we have discussed here we will be revisiting again in more detail later on in the book.

The essential points we have covered in this chapter are:

- Java applets are programs that are designed to be embedded in an HTML document. Java applications are standalone programs. Java applications can be console programs that only support text output to the screen, or they can be windowed applications with a GUI.
- Java programs are intrinsically object-oriented.
- Java sourcecode is stored in files with the extension `.java`.
- Java programs are compiled to byte codes, which are instructions for the Java Virtual Machine. The Java Virtual Machine is the same on all the computers on which it is implemented, thus ensuring the portability of Java programs.
- Java object code is stored in files with the extension `.class`.
- Java programs are executed by the Java interpreter, which analyses the byte codes and carries out the operations they specify.
- The Java System Development Kit ( the SDK) supports the compilation and execution of Java applications and applets.
- Experience is what you get when you are expecting something else.

# Resources

You can download the sourcecode for the examples in the book from any of:

- http://www.wrox.com
- ftp://www.wrox.com
- ftp://www.wrox.co.uk

The sourcecode download also includes ancillary files, such as `.gif` files containing icons for instance, where they are used in the examples.

If you have any questions on the fine formal detail of Java, the reference works we've used are:

- *The Java Language Specification, Second Edition (The Java Series)* James Gosling et al., Addison-Wesley, ISBN 0-201-31008-2
- *The Java Virtual Machine Specification Second Edition* Tim Lindholm and Frank Yellin, Addison-Wesley, ISBN 0-201-43294-3,

Other sites of interest are:

- http://www.wrox.com for support for this book and information on forthcoming Java books.
- http://p2p.wrox.com for lists where you can get answers to your Java problems.
- http://java.sun.com/docs/books/tutorial/index.html for the JavaSoft tutorials. Follow that Java trail.

and for online magazine reading and opinion, check out:

- http://www.javaworld.com/javasoft.index.html
- http://www.javareport.com/
- http://www.sys-con.com/java/

We also like the Java Developer Connection, subscribe to it at http://java.sun.com/jdc