

A Beginners Introduction to Java 2

August 4, 2000

Copyright © 2000, by James W. Cooper

IBM Thomas J Watson Research Center

Contents

Copyright © 2000, by James W. Cooper	1
IBM Thomas J Watson Research Center	1
Contents	2
1. HOW A COMPUTER WORKS	19
Computer Memory.....	19
Computer Programs	20
Making Decisions	21
Compilers	22
Java Compilers.....	22
Why Write Computer Programs?.....	23
2. WHAT IS JAVA?	25
The Genesis of Java	25
Executing Java Programs on Real Computers	25
Java Applets	26
Java Applications.....	27
Advantages of Java.....	28
Versions of Java	29
Summary	29
3. INSTALLING AND USING JAVA	30

Installing the JDK.....	30
Modifying Your Path	31
Java Program Files.....	32
Writing Java Programs	32
Compiling Java Programs	32
Applets vs Applications	33
Performance and Just-In-Time Compilers.....	36
Summary	36
4. SYNTAX OF THE JAVA LANGUAGE	37
Declaring Variables.....	38
Data Types	38
Numeric Constants	40
Character Constants	40
Variables	41
Declaring Variables as You Use Them.....	41
Multiple Equals Signs for Initialization.....	42
A Simple Java Program	42
Compiling and Running this Program.....	44
Arithmetic Operators	44
Increment and Decrement Operators	45
Combined Arithmetic and Assignment Statements	45
Making Decisions in Java	45

Comparison Operators	47
Combining Conditions	47
The Most Common Mistake	48
The switch Statement	48
Java Comments	49
The Confusing Ternary Operator	49
Looping Statements in Java	50
The For Loop	51
Declaring Variables as Needed in For Loops	51
Commas in For Loop Statements	52
How Java Differs from C	52
Summary	53
5. USING PROGRAMMING EDITORS TO WRITE JAVA PROGRAMS	54
Using Kawa	54
Using Visual SlickEdit	57
Summary	62
6. OBJECT ORIENTED PROGRAMMING	63
A Procedural Program	64
An Object-Oriented Program	64
Creating Instances of Objects	65
Constructors	66

A Java Rectangle Drawing Program.....	66
The Main Rect1 Program.....	67
Constructors.....	69
Methods inside Objects	69
Syntax of Methods	70
Variables	70
Capitalization Conventions in Java	71
Multiple Methods with the Same Name	71
Passing Arguments by Value	72
Objects and References	73
Object Oriented Jargon	75
Summary.....	75
7. USING CLASSES IN JAVA PROGRAMMING.....	77
The String Class.....	77
String Constructors.....	77
String Methods	78
The String +-operator	78
Conversion of Numbers to Strings and Vice-versa.....	79
Changing String Contents	80
The StringTokenizer	80
The Array Class.....	81
Garbage Collection.....	82

Vectors	82
Hashtables and Hashmaps	83
Constants in Java	83
Summary	84
8. INHERITANCE	85
Drawing a Square	87
Inheritance Terminology.....	89
The instanceof Operator	89
Overriding Methods in Your Derived Class.....	90
Understanding public, private and protected.....	92
Inheriting “Event Methods”	93
Abstract Classes	94
Interfaces.....	95
Interfaces Used in the Java Classes	96
The Enumeration Interface.....	96
The Iterator Interface	96
Summary	97
9. JAVA VISUAL CONTROLS.....	98
The Visual Class Hierarchy	99
Event Driven Programming	101
Event-Driven Programming	102
Action Events	104

The Controls Demonstration Program	105
The Button Control	106
The Label	107
TextFields & TextAreas	107
The List box.....	108
The Choice Box.....	109
The Scroll Bar	110
Checkboxes	111
Radio Buttons	112
Colors	113
Fonts	113
FontMetrics	114
Summary	114
10. INTRODUCTION TO THE SWING CLASSES	117
Installing and Using the JFC	117
Ideas Behind Swing	117
When to Use Swing Classes.....	118
The Swing Class Hierarchy.....	118
Writing a Simple JFC Program.....	119
Setting the Look and Feel	119
Setting the Window Close Box.....	120
Making a JFrame Class.....	120

A Simple Two Button Program	120
JButtons.....	122
RadioButtons and Toolbars	123
Radio Buttons	123
The JToolBar	124
JToggleButton.....	124
Sample Code	125
Borders in Swing	125
The JList Class.....	127
Summary	128

11. WRITING A SIMPLE VISUAL APPLICATION IN JAVA 130

Windows in Java	130
Building a Temperature Conversion Applet.....	130
Laying out Controls on the Screen.....	131
Applet Methods	131
The init() Method.....	132
The start() and stop() Methods	132
Laying Out Our Applet.....	132
Creating Objects	133
Initializing the Controls	133
Adding the Buttons and an ActionListener	134

The clickedCompute Method	135
Applets vs. Applications	138
Stopping Your Application from the System Close Button	139
Making an Application/Applet Exit	140
Subclassing the Button Control	141
Summary	143
12. LAYOUT MANAGERS.....	144
Java Layout Managers	144
The Flow Layout.....	145
The Grid Layout	148
The BorderLayout Manager.....	150
Padding a Layout Using Panels	152
Laying Out the Temperature Conversion Program	153
The CardLayout Manager	157
The GridBagLayout Manager.....	157
A Simple GridBagLayout Example	159
Improving on Our Simple GridBagLayout.....	161
A GridBagLayout for the TempCalc Program	163
Summary	166
13. WRITING A PROGRAM WITH TWO WINDOWS	167
A Temperature Plotting Program.....	167
The Plotit Display	171

Summary	173
14. FILES AND EXCEPTIONS.....	174
The File Class.....	174
Exceptions	175
Kinds of Exceptions	176
Reading Text Files	177
Multiple Tests in a Single Try Block	179
Catching More Than One Exception	180
Building an InputFile Class	182
Building an OutputFile Class.....	186
PrintStream Methods	187
Using the File Class.....	190
File Methods	191
Moving Through Directories	192
The FilenameFilter Interface.....	194
File Selection	196
Summary	197
15. USING THREADS IN JAVA.....	199
The Thread Object.....	199
Making an Object Into a Thread.....	200
Reasons for Using Threads	201

Adding Threads to this Program.....	203
Another Syntax for Starting a Thread.....	206
Several Threads in the Same PROGRAM.....	207
Building a Timer Class.....	210
The TimerBox Program.....	211
Synchronized Methods	214
The Model Town Bank	214
A Bureaucratic Solution	219
Objects in the Bank	220
Synchronizing Classes You Can't Change.....	222
Other Thread Properties	223
Daemon Threads	223
Thread Priorities	223
Thread Groups	224
Summary	224
16. IMAGES IN JAVA APPLETS AND APPLICATIONS..	225
Displaying an Image in an Applet.....	226
Displaying an Image in an Application.....	227
Using the MediaTracker Class.....	228
Converting Between Image Types.....	231
Making Blinking Alternating Images.....	232
The ImageIcon Class.....	233

Summary	234
17. MENUS AND DIALOGS	235
How Menus are Used.....	235
Menu Conventions	236
Creating Menus	236
MenuItem Methods	238
Creating Submenus	238
Other Menu Features	239
Checkbox Menus.....	239
Receiving Menu Commands	240
Swing Menus and Actions	240
Action Objects	241
Dialogs.....	244
The File Dialog.....	244
The Dialog Class.....	245
Calling Dialogs from Applets	247
Summary	249
18. USING THE MOUSE IN JAVA	251
Changing the Mouse Cursor Type	252
The MouseAdapter Class	253
Changing the Cursor in Separated Controls.....	254

Capturing Mouse Clicks	256
Double Clicking	257
Double Clicking in List Boxes	258
Dragging with the Mouse.....	258
Limitations on Dragging	259
Right and Center Button Mouse Clicks.....	260
Summary	262
19. ADVANCED SWING CONCEPTS.....	263
The JList Class	263
List Selections and Events	263
Changing a List Display Dynamically.....	264
Building an Adapter to Emulate the Awt List Class.....	266
The Adapter.....	267
A Sorted JList with a ListModel.....	269
Sorting More Complicated Objects	271
Getting Database Keys	273
Pictures in our List Boxes	275
The JTable Class.....	276
A Simple JTable Program.....	276
Cell Renderers	280
Rendering Other Kinds of Classes	282
Selecting Cells in a Table	284

Programs on the CD-ROM	285
The JTree Class.....	286
The TreeModel Interface.....	288
Programs on the CD-ROM.....	288
Summary	288
20. USING PACKAGES	290
The Java Packages	290
The Default Package	291
How Java Finds Packages	291
The Extension Folder	292
Jar Files	292
Creating Your Own Package.....	293
The package Statement	294
An Example Package	294
Class Visibility within Packages	295
Making Your Own Jar Files	295
Using a Jar File.....	297
Summary	297
21. WRITING BETTER OBJECT ORIENTED PROGRAMS	
298	
A Simple Implementation.....	299
Taking Command.....	301

Making Better Choice.....	302
Mediating the Final Difference	304
Summary	306
22. BUILDING WEB PAGES.....	308
HTML Documents.....	308
Creating a Web Page With a Word Processor	309
The Structure of an HTML Page	310
Headers	311
Line Breaks.....	312
Lists in HTML	313
Fonts and Colors	314
Background Colors	317
Images	317
Where to get Images.....	318
Putting Your Page on the Web	318
Hyper Links.....	319
Links to More Pages of Your Own	319
Links Within the Same Page.....	320
Mailto Links	321
Tables	322
Frames	324
Summary	325

23. WHAT IS JAVASCRIPT?.....	327
Differences Between Java and JavaScript	327
Embedding JavaScript Programs	328
JavaScript Variables.....	329
The JavaScript Objects on the Web Page.....	332
The Window Object.....	332
The History Object.....	334
The Document Object.....	334
Using Forms in JavaScript	335
The Button Control	335
A Simple Temperature Conversion Program.....	335
Properties of Form Controls in JavaScript	337
Validating User Input in JavaScript	338
Summary.....	340
24. INTERACTING WITH WEB PAGE FORMS	341
The HTML Form.....	341
Sending the Data to the Server	342
Form Fields	344
Selection Lists	344
Drop-down Lists.....	345
Making an Order Form.....	346

Submitting Your Form.....	349
Summary	350
25. USING SERVLETS.....	352
Why Bother with Servlets?	353
A Simple Servlet.....	353
Response to Fields in a Submitted Form.....	356
Ordering Dinner	357
Querying a Database.....	358
Writing the Servlet.....	359
Are There Really Two Servlets?.....	361
Multiple User Accesses to Your Servlet.....	364
Running Servlets on Your System.....	365
Summary.....	365
26. USING JAVA SERVER PAGES	367
Installing the JSP Development Kit.....	367
A Simple Hello World JSP program.....	368
The Java Bean and the JSP Page	369
Communicating with the Java Bean.....	370
Our First Complete JSP.....	371
Where to put all these classes?.....	373
Summary	373

27. MORE SOPHISTICATED JAVASERVER PAGES.....	375
A Names Factory.....	375
The JavaServer Page	377
More Class and Less Containment	378
Accessing Databases	379
The Database Bean	382
Conclusions.....	386

1. How a Computer Works

A computer is made up of a relatively simple collection of components. You will always have memory and disk drives for storing data and a processor for running programs. You will certainly also have a keyboard and mouse to communicate with the computer and a display and printer to get information back from the computer. You might also have a modem to dial up an internet service, or a network connection.

Computer Memory

Memory consists of individual *bits*, microscopic transistors which can hold a charge or not, and which we equate to ones and zeros. These bits are organized into groups of 8 bits, called *bytes*. When we say that a computer has 64 megabytes of memory, we mean that there are chips containing 64 million groups of 8 micro-transistors, or 512 million bits of memory.

Bytes can contain numbers from 0 to 2^8-1 or 0 to 255. You can see this in Table 1

<i>Table 1</i>	
<i>Values stored in 1 byte</i>	
Byte	Value
0000 0001	1
0000 0010	2
0000 0100	4
0000 1000	8
0100 0000	64
0111 1111	127
1000 0000	128
1111 1111	255

Bytes of course, can only hold rather small numbers, so they are usually grouped into *words* where a word might be 2, 4 or even 8 bytes. This corresponds to a computer that has 16, 32 or 64-bit words. At the time this book is being written, PCs commonly have a minimum of 32 megabytes (Mb) of memory, and often have 256 Mbytes, 512 Mbytes or even more memory. Today, most PCs utilize 32-bit words, but 64-bit computer prototypes are becoming more common.

Data in a computer's memory might take up one byte for each character of a string of text in a document, and 2 or 4 bytes for an integer value.

Computers can also store numbers in a *floating point* format, rather analogous to the exponential format we use to write large numbers, such as 6.02×10^{23} . Some bits are used for the exponent and some for the fractional part of the number. Floating point numbers usually reside in either 4 or 8 bytes or memory, for single or double precision numbers, respectively.

Computer memory is numbered in sequential bytes, and each number is referred to as the *address* of that byte. This is convenient because we can say that byte 1234 contains a specific value, say 31, and the next address contains some other value. When we write computer programs we frequently include addresses where we are to get data or store data.

Computer Programs

So far, we've only described how a computer holds numbers. In order for a computer to do useful work, it also has to be able to execute programs as well as storing data. So, some of the memory in a computer is designated as containing programs. Programs, however, are still just numbers, and which memory constitutes a program and which constitutes data is completely flexible and under the control of the programmer. Many times, when computers crash, it is because the computer begins executing data as if it were a program.

To see how computer programs work, let's consider a simple 4-bit program with just a few possible instructions:

Instruction	Operation
0000	Stop
0001	Fetch

0010	Add
0100	Subtract
1000	Store

We could then write a simple program to add two numbers together as

```
0001 Fetch
0010 Add
1000 Store
0000 Stop
```

Or, we could expand our program to include the memory addresses we want to get the data from and store the results in:

```
Fetch 1000      0001 1000
Add 1001        0010 1001
Store 1010     1000 1010
Stop           0000
```

So we see that a series of numbers stored in a computer's memory can be program instructions, data or a mixture of both. All that distinguishes them is which address we tell the computer to use to start executing instructions.

Making Decisions

If all computers could do was add and store numbers, they would be little more than fast adding machines, and not particularly significant. The important part of a computer is its ability to make decisions. While the computer instructions that make these decisions seem extremely elementary, in aggregate the ability to transfer control to another part of the program if a condition is true or false is extremely powerful.

Most computer decisions amount to making a test for some simple condition and then either jumping to a new address or not. Some typical decision instructions could be ones like

```
JmpIfZero
JmpIfNotZero
```

```

JumpIfNegative
JumpIfPositive

```

A program to take the absolute value of a number might be written as

```

    Fetch A
    JumpIfPositive B
    Negate
B: Store A

```

Statements like these are symbolic representations of the actual binary numbers that constitute the computer instructions the computer actually executes. These symbols have a one to one relationship to actual binary computer instructions and are translated into code like that in the binary example above using a program called an *assembler*.

Compilers

It is obvious that writing programs at this level can become very tedious indeed, although people did so in the early days of computers. Instead, we would prefer to write a program like

```

if (a < 0)
    a = -a;

```

This is a bit simpler and much easier to read. We need only write a program that translates statements like these into the machine language. Programs that make this somewhat larger leap are called *compilers*. There are many existing compilers for each of the many known computer languages. Languages like C, C++, Pascal and Visual Basic all have compilers that produce binary code for the target computer platform.

Java Compilers

Java is a high level language like C, C++ and Visual Basic, but Java compilers differ from most other compilers in that they produce compiled binary instructions not for the target computer hardware but for an abstract computer platform called the Java Virtual Machine. This Java machine does not actually exist in hardware. Instead the binary instructions are executed not by the processor of a computer, but by a computer program which *emulates* this theoretical machine. So, while the binary code a compiler generates for the C++ language will vary for Intel computers, Macintoshes

and Solaris-based computers, the binary codes produced by the Java compiler are *the same* on all 3 platforms. The program that interprets those binary statements is the only thing that varies. So, in Java, you can write a program and compile it on any platform, and feel confident that this compiled code will run without change on any other computer platform. This is the promise of “write once, run anywhere,” that makes Java such a persuasive language for development.

Of course, this portability comes at a price. You cannot expect a program which interprets the binary instructions to run as fast as an actual computer that executes these instructions directly. Java programs do run a bit more slowly than programs written in compiled languages. However, there have been a number of techniques applied to Java interpreters to make them more competitive in speed.

Java Just-In-Time compilers are the general solution to this problem. These compilers identify the spots in a program while it is running that can benefit by being optimized and carefully restructure the code or compile little pieces of it into machine language as it is executed, so that if that segment is run repetitively its speed will approach that of compiled languages. The first significant JIT compilers were from Symantec (now Webgain) and Microsoft. Microsoft seems to have abandoned this work, but the Symantec JIT compiler is widely available. More recently, both Sun and IBM have introduced JIT compiler systems which are quite powerful. Sun’s is referred to as its Hot Spot compiler, and is provided with the Windows and Solaris versions of Java 1.3.

Why Write Computer Programs?

If you are new to computing, you might ask why people do this? Why do they spend so much time writing programs for computers? We asked a lot of programmers of various kinds and ages why they write programs. Here are some of the answers they gave us.

1. To build really neat web sites.
2. To perform tasks that are boring and repetitive.
3. To analyze large amounts of data we have accumulated from studies or experiments.
4. To search for similarities in collections of data.

5. To build friendly user interfaces for programs that carry out computations, so other people can use them easily.

Some of these problems may sound familiar to you. There may be some other reasons you can think of too. Think about why you might want to write or use computer programs, and keep these reasons in mind as you begin to learn about the powerful Java language.

2. What Is Java?

The Genesis of Java

The Java language was designed at Sun Microsystems over a five year period, culminating in its first release in January 1996. The developers, led by James Gosling, originally set out to build a language for controlling simple home devices with embedding microcomputers. Thus, they started with the object-oriented concepts of C++, simplified it, and removed some of the features, such as pointers, that lead to serious programming errors.

While the team may have originally had in mind compiling this language for a specific microprocessor, instead they developed both the Java language and a hypothetical processor called the Java Virtual Machine or JVM. The Java compilers they developed produced binary code designed to execute on the JVM rather than on a PC or Sun workstation.

The team's working name for this evolving language was "Oak," but as it neared completion, they found that Oak was already registered as a trademark. As the story goes, they came up with the name "Java" while taking a break at a coffee shop.

Executing Java Programs on Real Computers

To actually execute Java programs, they developed Java *interpreters* that ran on various machines and under various operating systems. Thus, Java became a language that would execute on a number of systems and now has implementations for virtually all common computers and operating systems.

Sometime during development, it suddenly became obvious that Java would be an ideal language for use on the Internet, the popularity of which was growing at a phenomenal rate.

They added a window manager to allow easy development of user interfaces; they also added network communication methods such as Web page URLs and sockets.

Since Java could now run on nearly any kind of workstation, it became an ideal vehicle for adding powerful computational capabilities to Web pages.

You could browse Web pages from any platform and see the same behavior in the embedded Java program.

Java Applets

A Java *applet* is a program designed to be embedded in a Web page. Applets can be quite complex; they are not limited to simple animations or single windows. They can access remote databases or other sources of information and load and operate on complex information on your computer system.

Java applets are designed to be quite *secure*, however. They can be safely downloaded over a network without causing concern that they might be able to do damage or mischief to your computer.

They cannot write any information onto your hard disk unless you specifically give them access to a directory. They cannot access any other resources on your network or any other hardware or peripherals on your computer.

Java applets are also restricted from writing into memory outside of the Java address space. This is accomplished mainly because Java has no memory pointer type and thus a malicious programmer cannot point to memory he or she might want to attack. Java applets are also scrutinized class by class as they are loaded by the run-time environment in your browser, and checks are made to assure that the binary code has not been modified such that it might interact with or change any part of the memory of your system.

Further, applets cannot access resources on any other computer on your network or elsewhere on the Internet, with one exception; they can open TCP/IP connections back to the machine running the Web server from which they were downloaded.

To add even more protection and prevent programmers from spoofing users into giving them confidential information, all windows that you pop up from an applet have a banner along the bottom reading “Unsigned Java Applet.” The banner serves to prevent hackers from designing an exact copy of a familiar screen and luring users to type confidential information into it.

Java Applications

By contrast, Java applications are full-featured programs that run on your computer and have full access to its resources. They can read, write, create, and delete files and access any other computer on the network.

You will quickly appreciate that it is possible to develop full-fledged applications in Java: database viewers, word processors, spreadsheets, math packages, and file and network manipulation programs. In fact, one of Java's great strengths is that it makes accessing other computers on your network extremely easy.

Now you can write quite sophisticated programs in many other languages, so you might ask whether Java is really "ready" for all this attention. Can you write real, significant programs in Java? Figure 1-1 shows a simple data entry program for a search tool that allows you to enter terms as well as Boolean conditions from drop-down list boxes created in Java.



Figure 1-1: A simple search query entry form.

The window shown in Figure 1-1 represent a complete, working program. If you want to see how it works, you'll find the sources and binaries on the Companion CD-ROM in the \chapter1 folder.

But what about this Java language? How does it really stack up? Is it real or is it a toy? Let's take a moment and look at some of its advantages.

Advantages of Java

The advantages of Java are substantial for both simple applications and for complex server code.

1. Java is object oriented. Java requires that you write 100 percent object-oriented code. As we will see, object-oriented (OO) programs are easier to write and easier to maintain than the spaghetti code that is often the result of programming in other languages.
2. Java works on most platforms. While C/C++ programs are platform specific, Java binary byte code runs identically on most Unix machines, Macintoshes, and PC's running Windows 95/98/ NT, Windows 2000, and Linux.
3. Java is network enabled. It is trivially simple to write code in Java that works across networks. The use of URLs, TCP sockets, and remote classes is essentially built into the language.
4. Java is multithreaded. You can write programs in which several sections run simultaneously in different execution threads.
5. Java allows you to add major function to Web pages. If you are interested in building interactive World Wide Web pages that compute, collect, or display data, Java is the language of choice. There simply is not a better way to add interactive controls to Web pages.

With all of these pluses, the only possible drawback is that you'll have to learn a new language. This is, of course, what this book is about and we'll see some very significant advantages to Java as we begin to explore it.

Versions of Java

When Java was first released in 1996, it was called Version 1.0. The following year, Java 1.1 was introduced which provided a more sophisticated and flexible way of processing user events. In 1999, Java 1.2 was released, and dubbed “Java 2.” Since then, additions to Java have focused on various additional packages for the enterprise, the microdevice and so forth. We will be discussing Java 2 in this book.

Summary

So far, we’ve looked at the reason Java was designed and compared it briefly to other languages. Next we’ll look at how to install Java and compile Java programs. Then, to become fluent in Java, you’ll need to learn two things: Java’s syntax and the rationale behind object-oriented programming. We’ll take these up in the following chapters and soon you, too, will be a Java expert.

3. Installing and Using Java

Installing the JDK

The Java Development Kit (JDK) is provided free of charge from Sun Microsystems (<http://java.sun.com>). You can download two files: the development kit library and executables and the documentation files. They are also provided on the accompanying CD-ROM. For Windows, these files are called JDK-13-win32.exe and JDK-13-apidocs.zip.

In this book, we will concentrate on the Windows environment because of its pervasiveness, but since Java runs anywhere, you will find that except for minor installation differences, your Java programs should run on most platforms. To install the JDK libraries, simply run the exe file:

```
JDK-13-win32
```

by double clicking on it.

This will create a `\jdk1.3` directory with a number of directories under it, notably `\jdk1.3\bin` and `\jdk1.3\lib`. It also installs the Java Runtime Environment (JRE) and the Java Plug-in which allows various browsers to utilize the current level of Java technology.

To install the documentation, you must unzip it using a zip program that preserves long filenames. WinZip from Nico-Mak Computing (<http://www.winzip.com>) is one such program. The old 16-bit Windows or DOS versions of PKZIP will not preserve the needed long filenames and you shouldn't try to use them.

Using WinZip, unpack the "apidocs" file into the directory `\java\api`. This produces a tree of small files containing descriptions of every class in the Java language. To view them, use your browser to open the `api\index.html` file. You will see a frame display like that in Figure 2-1 where you can look at all the classes and the methods of each library or *package* in the java language. Bookmark this page in your browser, because you will find it an invaluable reference in learning how to write useful Java programs.

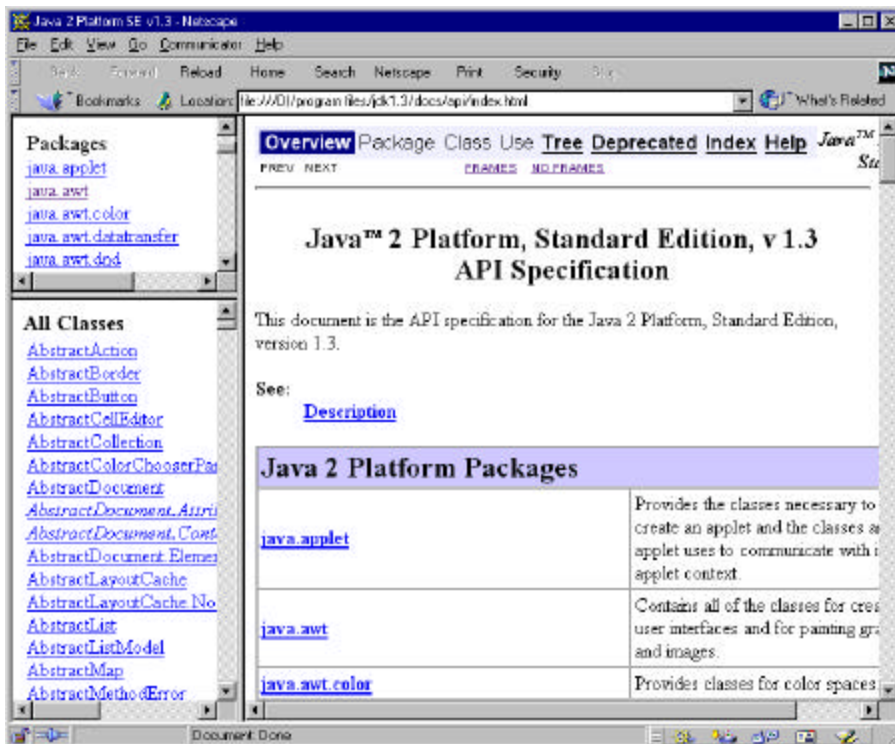


Figure 2-1 – The index of the api documents for the Java SDK.

Modifying Your Path

You need to add the `\java\bin` directory to your path so that the command line programs **java** and **javac** are available to you. Add the following statement to the bottom of your `autoexec.bat` file:

```
path=%path%;c:\Jdk1.3\bin;
```

if you have installed the programs on your C drive. For Windows NT and 2000, you should modify your Path by opening the System folder in the Control Panel.

You also may need to add the CLASSPATH environment variable to your system by inserting the following statement in your `autoexec.bat` file:

```
set CLASSPATH=.;c:\java\lib\tools.jar;
```

Note especially the “. ;” at the beginning of the CLASSPATH statement. This is required for Java programs to find the files that you compile in other directories. In Java 2, the CLASSPATH variable is much less important and can usually be omitted, if you make sure that other Java library files (Jar files) are installed in the \jre\lib\ext folder.

Java Program Files

All Java source files have the .java extension. For example the simulated search window program we showed in Chapter 1 is called srchapp.java . Note that since this is a 4-character extension, this qualifies as a long filename under Windows 95, and all of the tools you use for handling Java programs must be able to deal with long filenames.

When you compile a Java program, it produces one or more files having the “.class” extension which you then can operate on either directly with the **java** interpreter or with the **apple tvviewer** program.

Writing Java Programs

You can write Java programs with any text editor program provided with Windows 9x. You can also use the WordPad editor for this purpose. However, there are several editors that provide syntax highlighting and keyword completion which make Java development that much easier. We will be suggesting Visual SlickEdit and Kawa. The X2 editor is also freely available, at www.interlog.com/~bwt.

In addition, there are any number of integrated development environments which feature automatic indenting and syntax highlighting. These include Webgain Visual Café, Borland/Inprise JBuilder, IBM VisualAge for Java and several others. Note however, that whatever editing system you use, source code files for java programs always have the .java extension.

Compiling Java Programs

When you have written your first java program you compile it by running the **javac** compiler. Several of the integrated development environments do

this for you under the covers. As an example, let's compile the simple simulated search application we referred to in the last chapter. This program is in the \chapter1 directory of the example disk, and is called Srchapp.java. Before you can compile it, you must copy it to a directory on your disk where the compiler can read and write files. Then, to compile it, you type

```
javac Srchapp.java
```

Note that you must specify the program filename using the exact case of the long filename and that you must include the .java filename extension as part of the command line.

The java compiler will produce one or more output files having the .class extension. In this case, it generates the file

```
Srchapp.class
```

Then, to execute this program we run the Java interpreter, specifying the main file of the program:

```
java Srchapp
```

Note that here, the exact case of the filename is again required, but that the filename extension (.class) is *not* required. In fact, if you include the “.class” extension, it is an error. The Java interpreter runs the main program and searches for the other required class files in the path specified by the CLASSPATH environment variable. In this case, the interpreter looks first in the current directory, where the Srchapp.class program is located and finds the MainPanel1.class file as well. Then it looks in the tools.jar file in the c:\java\lib directory for any needed java support files.

Applets vs Applications

The Srchapp program we worked on above is a Java *application* or a stand-alone program which runs without respect to a web browser or web page. By contrast, *applets* are programs which are embedded in web pages and can only be run either by your browser or by a test program called **appletviewer**.

As we noted in the previous chapter, an applet is restricted in the access it has to your computer. It cannot read or write local files or environment variables and it cannot gain access to your network or to other computers than the computer providing the web server. To embed an applet in a web page, you need to include an `<applet>` tag in the HTML text of your web page. A simple web page which simply displays an applet is contained in the file `srchapp.html` which has the contents:

```
<html>
<body>
<applet code="srchapp.class"
        width = 400 height =300>
</applet>
</body>
</html>
```

The `srchapp` program we have been discussing has been written so that it can run either as an application or as an applet, so it will in fact work when embedded in a web page.

Then, if you want to view that applet, you can simply load that web page into your browser, or you can run the `appletviewer` program from your command line:

```
appletviewer srchapp.html
```

Note in particular that the target file for the `appletviewer` program is an HTML file, not a `.class` file. You can look at a similar search application embedded in a web page by typing the above command.

Figure 2-1 shows the program running as a stand-alone application using Java 1.1. We will see in later chapters that using the more professional-looking Java Foundation Classes (“Swing” classes) requires a more elaborate HTML page to tell the browser to load those classes, but that these pages can be created automatically.

Figure 2-2 shows the same application running using the Java 2 Swing classes and Figure 2-3 shows this same applet running on a web page in the Netscape browser.

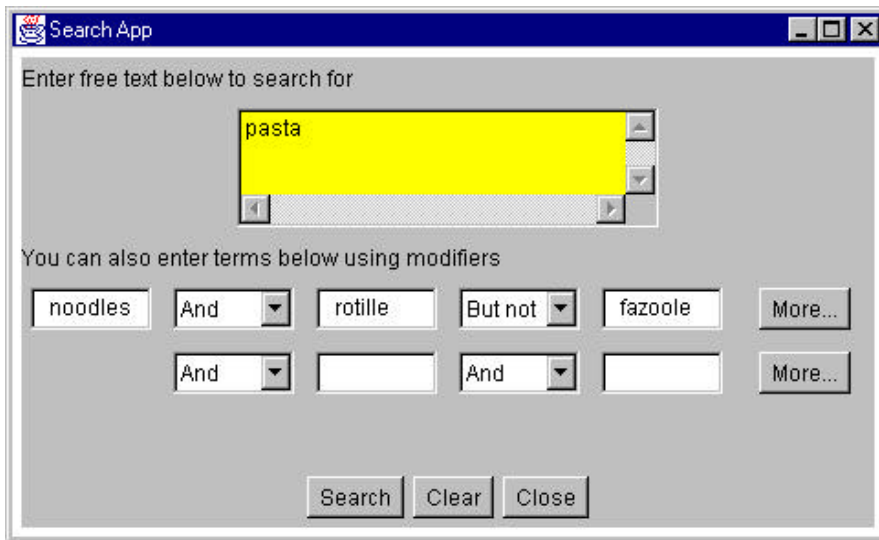


Figure 2-1. The simulated search application running using Java 1.1.

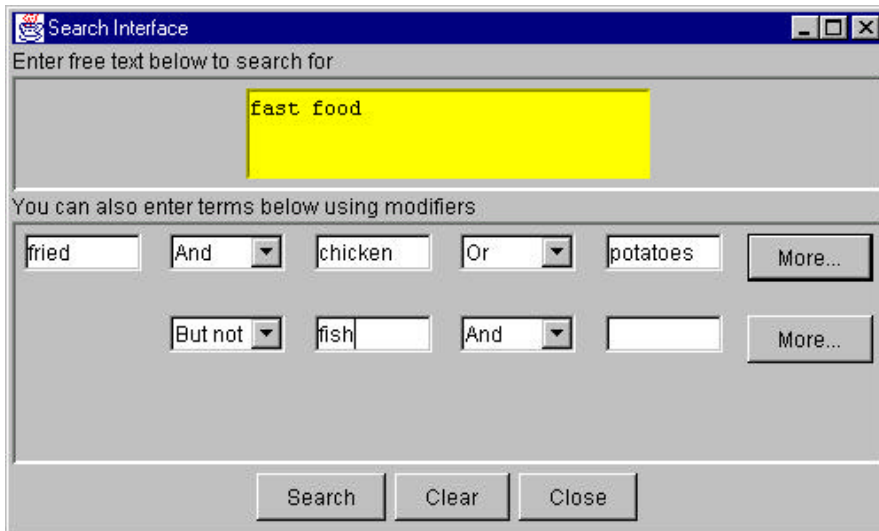


Figure 2-2: The simulated search application running using Java 2.

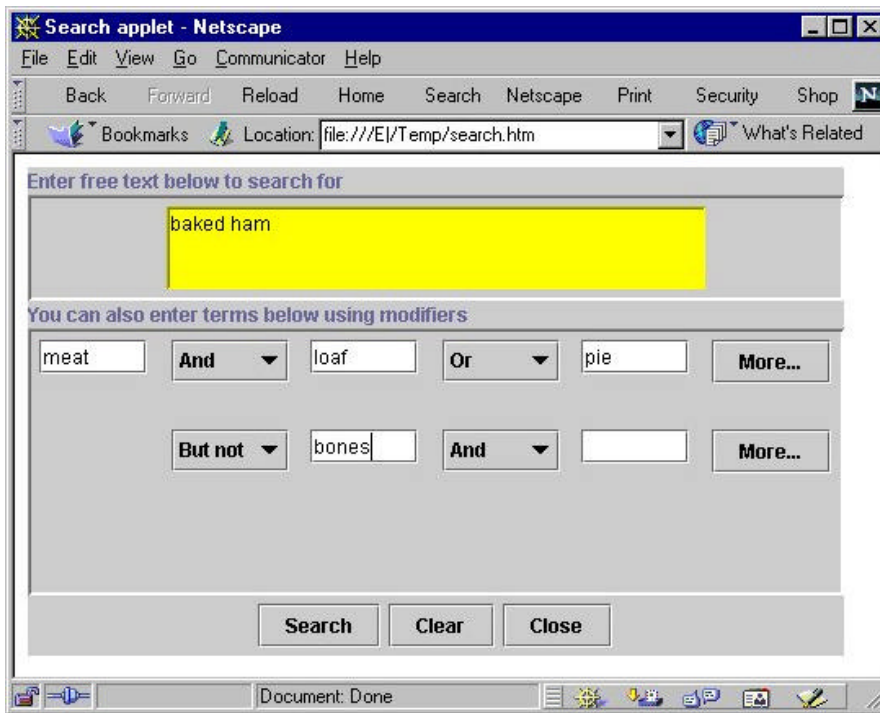


Figure 2-3: The simulated search application running inside Netscape.

Performance and Just-In-Time Compilers

Since Java is an interpreted language, it can sometimes be slower than compiled languages like C and C++. However for more computationally bound operations, Just-In-Time (JIT) compilers are now available. These JIT compilers interpret the byte codes as usual, but also translate them into local machine language so that if that code is executed more than once in a loop, all further executions will be executed as native machine instructions.

Summary

Now that we have a grasp of how Java is installed and how to compile simple programs, let's look at the language itself, and see how it compares with other related language.

4. Syntax of the Java Language

Java has all the features of any powerful, modern language. If you are familiar with C or C++ you will find most of its syntax very familiar. If you have been working in other languages or if Java is your first computer language, you should read this chapter. You'll quickly see that Java is a simple language with an easy-to-learn syntax.

The two major differences between Java and languages like Basic and Pascal are that Java is *case sensitive* (most of its syntax is written in *lower case*) and that every statement in Java is terminated with a semicolon(;). Thus, Java statements are not constrained to a single line and there is no line continuation character.

In Basic, we could write

```
y = m * x + b           'compute y for given x
```

or we could write

```
Y = M * X + b           'upper case Y, M and X
```

and both would be treated as the same. The variables Y, M and X are the same whether written in upper or lower case.

In Java, however, case is significant, and if we write

```
y = m * x + b;         //all lower case
```

or

```
Y = m * x + b;         //Y differs from y
```

we mean two different variables: “Y” and “y.” While this may seem awkward at first, having the ability to use case to make distinctions is sometimes very useful. For example, programmers often capitalize symbols referring to constants:

```
final float PI = 3.1416; // a constant
```

The **final** modifier in Java means that that named value is a constant and cannot be modified.

Programmers also sometimes define data types using mixed case and variables of that data type in lower case:

```
class Temperature    //begin definition of
                    //new data type
Temperature temp;   //temp is of this new type
```

We'll see much more about how we use classes in the chapters that follow.

Declaring Variables

Java requires that you *declare* every variable before you can use it. If you use a variable that you have not declared, the Java compiler will flag this as an error when you try to compile the program. You can declare the variables in several ways. Often, you declare several at the same time.

```
int y, m, x, b;           //all at once
or one at a time.
```

```
int y;                   //one at a time
int m;
int x;
int b;
```

More commonly, in Java you declare the variables as you use them.

```
int x = 4;
int m = 8;
int b = -2;

int y = m * x + b; //in an expression
```

However, you must declare variables by the time you first refer to them.

Data Types

The major data types in Java are shown in the table below:

Type	contents
boolean	True or false
byte	signed 8-bit value
short	16-bit integer
int	32-bit integer
long	64-bit integer
float	32-bit floating point
double	64-bit floating point
char	16-bit character
String	16-bit characters

Note that the lengths of these basic types are irrespective of the computer type or operating system. This differs from C and C++, where the length of an integer and a long can vary with the computer platform.

Characters and Strings in Java always use 16-bit characters: to allow for representation of characters in non-Latin languages. Java uses a character coding system called Unicode.

You can convert between variable types in the usual simple ways.

- Any “wider” data type can have a lower data type assigned directly to it and the promotion to the new type will occur automatically. If *y* is of type **float** and *j* is of type **int** then you can write:

```
float y;      //y is of type float
int j; //j is of type int
y = j; //convert int to float
```

to promote an integer to a float.

- You can reduce a wider type to a narrower type using by *casting* it. You do this by putting the data type name in parentheses and putting this name in front of the value you wish to convert:

```
j = (int)y; //convert float to integer
```

Boolean variables can only take on the values represented by the reserved words **true** and **false**. Boolean variables also commonly receive values as a result of comparisons and other logical operations:

```
int k;
boolean gtnum;

gtnum = (k > 6);    //true if k is greater than 6
```

Unlike C, you cannot assign numeric values to a boolean variable and you cannot convert between **boolean** and any other type.

Numeric Constants

Any number you type into your program is automatically of type **int** if it has no fractional part or of type **double** if it does. If you want to indicate that it is a different type, you can use various suffix and prefix characters to indicate what you had in mind.

```
float loan = 1.23f; //float
long pig   = 45L;   //long
long color = 0x12345; //hexadecimal
int register = 03744; //octal: leading zero
```

Java also has three reserved word constants: **true**, **false**, and **null**, where **null** means an object variable which does not yet refer to any object. We'll learn more about objects in the next chapters.

Character Constants

You can represent individual characters by enclosing them in single quotes:

```
char c = 'q';
```

Java follows the C convention that the “whitespace characters” can be represented by preceding special characters with a backslash. Since the backslash itself is thus a special character, it can be represented by using a double backslash.

'\n'	newline (line feed)
'\r'	carriage return

'\t'	tab character
'\b'	backspace
'\f'	form feed
'\0'	null character
'\"'	double quote
'\''	single quote
'\\'	backslash

Variables

Variable names in Java can be of any length and can be of any combination of upper and lower case letters and numbers, but the first character must be a letter. Further, since Java uses Unicode representations throughout, you can intermingle characters from other language fonts if you wish, but this is usually more confusing than it is useful.

```
PI = 3.1416;
```

Note that since case is significant in Java, the following variable names all refer to different variables:

```
temperature
Temperature
TEMPERATURE
```

Declaring Variables as You Use Them

Java also allows you to declare variables just as you need them rather than requiring that they be declared at the top of a procedure:

```
int k = 5;
float x = k + 3 * y;
```

This style is very common in object-oriented programming, where we might declare a variable inside a loop which has no existence or *scope* outside that local spot in the program.

Multiple Equals Signs for Initialization

Java, like C, allows you to initialize a series of variables to the same value in a single statement:

```
i = j = k = 0;
```

This can be confusing to read, so do not overuse this feature. The compiler will generate the same code for the above statement as for:

```
i = 0; j = 0; k = 0;
```

whether the statements are on the same or successive lines.

A Simple Java Program

Now let's look at a very simple Java program for adding two numbers together. This program is a stand-alone program or *application*. We will also see later that Java *applets* have a similar style in many ways, but do not require a **main** function.

```
import java.io.*
public class add2 {
    public static void main(String arg[]) {
        double a, b, c;
        a = 1.75;
        b = 3.46;
        c = a + b;
        //print out sum
        System.out.println("sum = " + c);
    }
}
```

This is a complete program as it stands, and if you compile it with the **javac** compiler and run it with the **java** interpreter, it will print out the result:

```
sum = 5.21
```

Let's see what observations we can make about this simple program:

1. You must use the **import** statement to define libraries or *packages* of Java code that you want to use in your program. This is similar to the C and C++ **#include** directive. (The two packages we refer to in the example above are for illustration only. This simple program does not actually need them.)
2. The program starts from a function called **main** and it must have *exactly* the form shown here:

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

3. Every program module must contain one or more classes.
4. The class and each function within the class is surrounded by *braces* (`{ }`). Leaving out a closing brace or duplicating the final brace are common errors. Many development environments try to help you keep your braces paired so the compiler does not get confused.
5. Every variable must be declared by type before or by the time it is used. You could just as well have written:

```
double a = 1.75;  
double b = 3.46;  
double c = a + b;
```

6. Every statement must terminate with a semicolon. They can go on for several lines if you want but must terminate with the semicolon.
7. Comments start with `“//”` and terminate at the end of the line.
8. Like most other languages (except Pascal) the equals sign is used to represent assignment of data
9. You can use the `+`-sign to combine two strings. The string `“sum =”` is concatenated with the string representation of the double precision variable `c`.
10. You use the `System.out.println` function to print values on the screen.

Compiling and Running this Program

This simple program is called “add2.java” in the \javalang directory on your example disk. You can compile and execute it by copying it to any convenient directory and typing

```
javac add2.java
```

and you can execute it by typing

```
java add2
```

Arithmetic Operators

The fundamental operators in Java are much the same as they are in most other modern languages.

Java	
+	Addition
-	subtraction, unary minus
*	Multiplication
/	Division
%	modulo (remainder after integer division)

The bitwise and logical operators are derived from C. They all return numerical values.

Java	
&	bitwise And
	bitwise Or
^	bitwise exclusive Or
~	one's complement
>> <i>n</i>	right shift <i>n</i> places
<< <i>n</i>	left shift <i>n</i> places

Increment and Decrement Operators

Like C/C++ and completely unlike other languages Java allows you to express incrementing and decrementing of integer variables using the “++” and “--” operators. You can apply these to the variable before or after you use it:

```
i = 5;
j = 10;
x = i++;      //x = 5, then i = 6
y = --j;     //y = 9 and j = 9
z = ++i;     //z = 7 and i = 7
```

Combined Arithmetic and Assignment Statements

Java allows you to combine addition, subtraction, multiplication and division with the assignment of the result to a new variable.

```
x = x + 3;           //can also be written as:
x += 3;             //add 3 to x; store result in x

//also with the other basic operations:
temp *= 1.80;      //mult temp by 1.80
z -= 7;           //subtract 7 from z
y /= 1.3;         //divide y by 1.3
```

This is used primarily to save typing: it is unlikely to generate any different code. Of course, these compound operators (as well as the ++ and – operators) cannot have spaces between them.

Making Decisions in Java

The familiar if-then-else of Visual Basic, Pascal and Fortran has its analog in Java. Note that in Java, however, we do not use the “then” keyword.

```
if ( y > 0 )
    z = x / y;
```

Parentheses around the condition are *required* in Java. This format can be somewhat deceptive: as written only the single statement following the **if** is operated on by the if-statement. If you want to have several statements as part of the condition, you must enclose them in braces:

```
if ( y > 0 ) {
```

```

z = x / y;
System.out.println("z = " + z);
}

```

By contrast, if you write

```

if ( y > 0 )
    z = x / y;
System.out.println("z = " + z);

```

the Java program will always print out “z=” and some number, because the **if** clause only operates on the single statement that follows. As you can see, indenting does not affect the program: it does what you say, not what you mean.

If you want to carry out either one set of statements or another depending on a single condition, you should use the **else** clause along with the **if** statement.

```

if ( y > 0 )
    z = x / y;
else
    z = 0;

```

and if the else clause contains multiple statements, they must be enclosed in braces as above.

There are two or more widely used indentation styles for braces in Java programs: that shown above will be familiar to Pascal programmers. The other style, popular among C programmers places the brace at the end of the **if** statement and the ending brace directly under the if:

```

if ( y > 0 ) {
    z = x / y;
    System.out.println("z=" + z);
}

```

You will see both styles widely used.

Comparison Operators

Above, we used the “>” operator to mean “greater than.” Most of these operators are the same as in other languages. Note particularly that *is equal to* requires *two* equals signs and that “not equal” is an exclamation point followed by an equals sign.

Java	Meaning
>	greater than
<	less than
==	is equal to
!=	is not equal to
>=	greater than or equal to
<=	less than or equal to
!	not

All of these operators return boolean results.

Combining Conditions

When you need to combine two or more conditions in a single **if** or other logical statement, you use the symbols for the logical and, or, and not operators. These are totally different than from any other languages except C/C++ and are confusingly like the bitwise operators shown above:

Java	
&&	logical And
	logical Or
~	logical Not

So, in Java we would write

```
if ( (0 < x) && ( x <= 24) )
    System.out.println ("Time is up");
```

The Most Common Mistake

Since the “is equal to” operator is “==” and the assignment operator is “=” they can easily be misused. If you write

```
if (x = 0)
    System.out.println("x is zero");
```

instead of

```
if (x == 0)
    System.out.println("x is zero");
```

you will get the odd-seeming compilation error, “Cannot convert double to boolean,” because the result of the fragment

```
(x = 0)
```

is the double precision number 0, rather than a boolean **true** or **false**. Of course, the result of the fragment

```
(x == 0)
```

is indeed a boolean quantity and the compiler does not print any error message.

The switch Statement

The **switch** statement lets you provide a list of possible values for a variable and code to execute if each is true. However, in Java, the variable you compare in a **switch** statement must be either an integer or a character type and must be enclosed in parentheses:

```
switch ( j ) {
    case 12:
        System.out.println("Noon");
        break;
    case 13:
        System.out.println("1 PM");
        break;
    default:
        System.out.println("some other time...");
```



```
}

```

Note particularly the **break** statement following each case in the switch statement. This is very important in Java as it says “go to the end of the switch statement.” If you leave out the **break** statement, the code in the next case statement is executed as well.

Java Comments

As you have already seen, comments in Java start with a double forward slash and continue to the end of the current line. Java also recognizes C-style comments which begin with “/*” and continue through any number of lines until the “*/” symbols are found.

```
//Java single-line comment
/*other Java comment style*/
/* also can go on
for any number of lines*/

```

You can’t nest Java comments, once a comment begins in one style it continues until that style concludes.

Your initial reaction as you are learning a new language may be to ignore comments, but they are just as important at the outset as they are later. A program never gets commented at all unless you do it as you write it, and if you want to ever use that code again, you will find it very helpful to have some comments to help you in deciphering what you meant for it to do. Many programming instructors refuse to accept programs that are not thoroughly commented for this reason.

The Confusing Ternary Operator

Java has unfortunately inherited one of C/C++’s most opaque constructions, the ternary operator. The statement

```
if ( a > b )
    z = a;
else
    z = b;

```

can be written extremely compactly as

```
z = (a > b) ? a : b;
```

The reason for the original introduction of this statement into the C language was, like the post-increment operators, to give hints to the compiler to allow it to produce more efficient code. Today, modern compilers produce identical code for both forms given above, and the necessity for this turgidity is long gone. Some C programmers coming to Java find this an “elegant” abbreviation, but we don’t agree and will not be using it in this book.

Looping Statements in Java

Java has only 3 looping statements: **while**, **do-while**, and **for**. In Java we write

```
i = 0;
while ( i < 100)  {
    x = x + i++;
}
```

The loop is executed as long as the condition in parentheses is true. It is possible that such a loop may never be executed at all, and, of course, if you are not careful, that a while loop will never be completed.

The Java **do-while** statement is quite analogous, except that in this case the loop must always be executed at least once, since the test is at the bottom of the loop:

```
i = 0;
do {
    x += i++;
}
while (i < 100);
```

The For Loop

The **for** loop provides a way for you to pass through code a specific number of times. It has three parts, an initializer, a condition and an operation that takes place each time through the loop, each separated by semicolons.

```
for (i = 0; i < 100; i++) {
    x += i;
}
```

Let's take this statement apart:

```
for (i = 0;           //initialize i to 0
     i < 100 ; //continue as long as i < 100
     i++)           //increment i after every pass
```

In the loop above, **i** starts the first pass through the loop set to zero. A test is made to make sure that **i** is less than 100 and then the loop is executed. After the execution of the loop the program returns to the top, increments **i** and again tests to see if it is less than 100. If it is, the loop is again executed.

Note that this **for** loop carries out exactly the same operations as the **while** loop illustrated above. It may never be executed and it is possible to write **for** loops that never exit.

Declaring Variables as Needed in For Loops

One very common place to declare variables on the spot is when you need an iterator variable for a **for** loop. You simply declare that variable right in the **for** statement as follows:

```
for (int i = 0; i < 100; i++)
```

Such a loop variable exists or has *scope* only within the loop. It vanishes once the loop is complete. This is important because any attempt to reference such a variable once the loop is complete will lead to a compiler error message. The following code is incorrect:

```
for (int i =0; i < 5; i++)
    x[i] = i;
```

```
//the following statement is in error
//because i is now out of scope
System.out.println("i=" + i);
```

Commas in For Loop Statements

You can initialize more than one variable in the initializer section of the Java **for** statement, and you can carry out more than one operation in the operation section of the statement. You separate these statements with commas:

```
for (x=0, y= 0, i =0; i < 100; i++, y +=2) {
    x = i + y;
}
```

It has no effect on the loop's efficiency, and it is far clearer to write

```
x = 0;
y = 0;
for ( i = 0; i < 100; i++) {
    x = i + y;
    y += 2;
}
```

It is possible to write entire programs inside an over-stuffed **for** statement using these comma operators, but this is only a way of obfuscating the intent of your program.

How Java Differs from C

If you have been exposed to C, or if you are an experienced C programmer, you might like to understand the main differences between Java and C. Otherwise you can skip this section.

1. Java does not have *pointers*. There is no way to use, increment or decrement a variable as if it were an actual memory pointer.
2. You can declare variables anywhere inside a method you want to: they don't have to be at the beginning of the method.
3. You don't have to declare an *object* before you use it: you can define it later.

4. Java does not have the C **struct** or **union** types. You can carry out most of these features using classes. It also does not support **typedef** which is commonly used with **structs**.
5. Java does not have enumerated types, which allow a series of named values to be assigned sequential numbers, such as colors or day names.
6. Java does not have bit fields: variables that take up less than a byte of storage.
7. Java does not allow variable length argument lists. You have to define a method for each number and type of arguments.

Summary

In this brief chapter, we have seen the fundamental syntax elements of the Java language. Now that we understand the tools, we need to see how to use them. In the chapters that follow, we will take up objects and show how to use them and how powerful they can be .

5. Using Programming Editors to Write Java Programs

While it is possible to write all your Java programs using an ordinary editor like the Wordpad editor provided with Windows, you can do a lot better than that using one of a number of popular programming tools. Two of these we'll discuss here are Kawa and Visual SlickEdit.

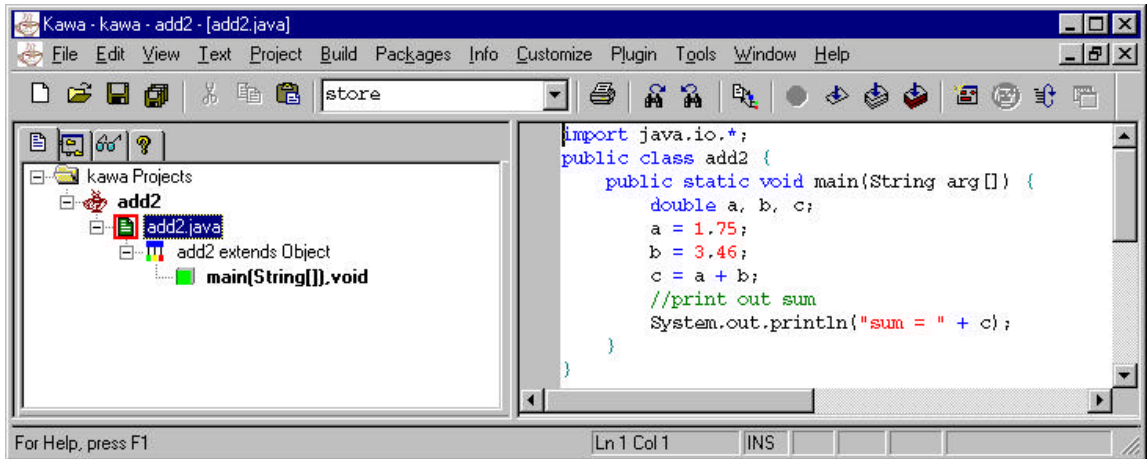
The Kawa Java programming editor is itself written in Java and provides an integrated environment where you can write, compile, execute and debug Java programs. The editor features syntax highlighting and code beautification, as well as management of multiple file projects and source code control. You can download an evaluation copy from www.tek-tools.com.

Visual SlickEdit is a programmer's editor, which is configurable for a number of languages, and editing styles. While, by default, it behaves like any Windows editor, you can configure it to work like Emacs, if you are familiar with that popular Unix-based editor. In addition to syntax highlighting, project file management, syntax highlighting, and a compile and build environment, it also provides keyword completion for any Java class methods you use. It does not, however, provide any Java debugging or source code control.

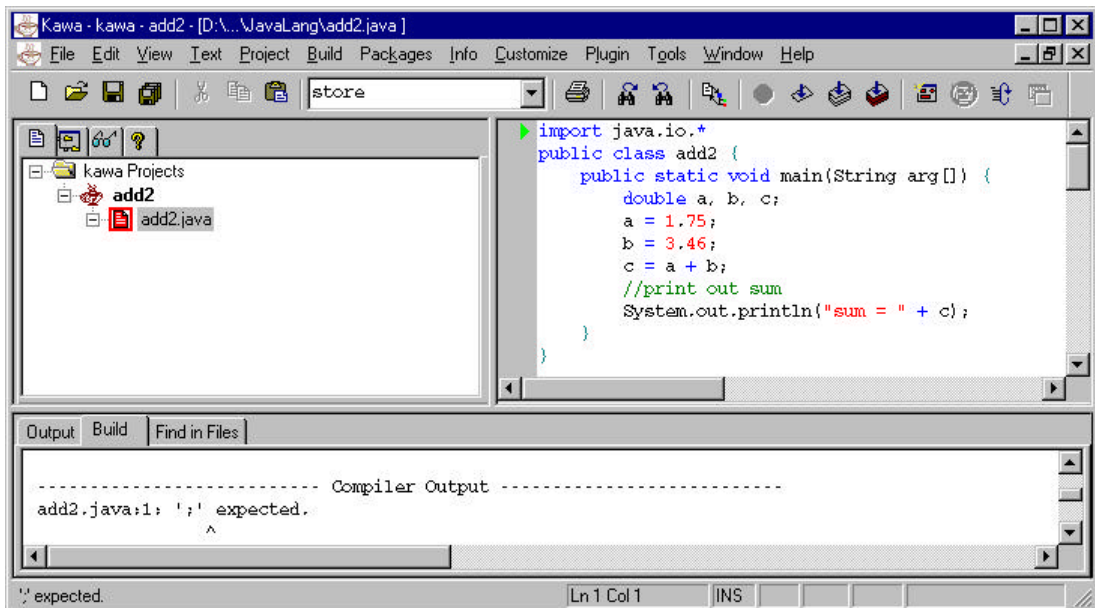
These editor systems are quite desirable, because Java programs tend to consist of a large number of files: one for each class in the program, and these tools help you keep track of all these files and make sure that all of them are recompiled when you change any one of them.

Using Kawa

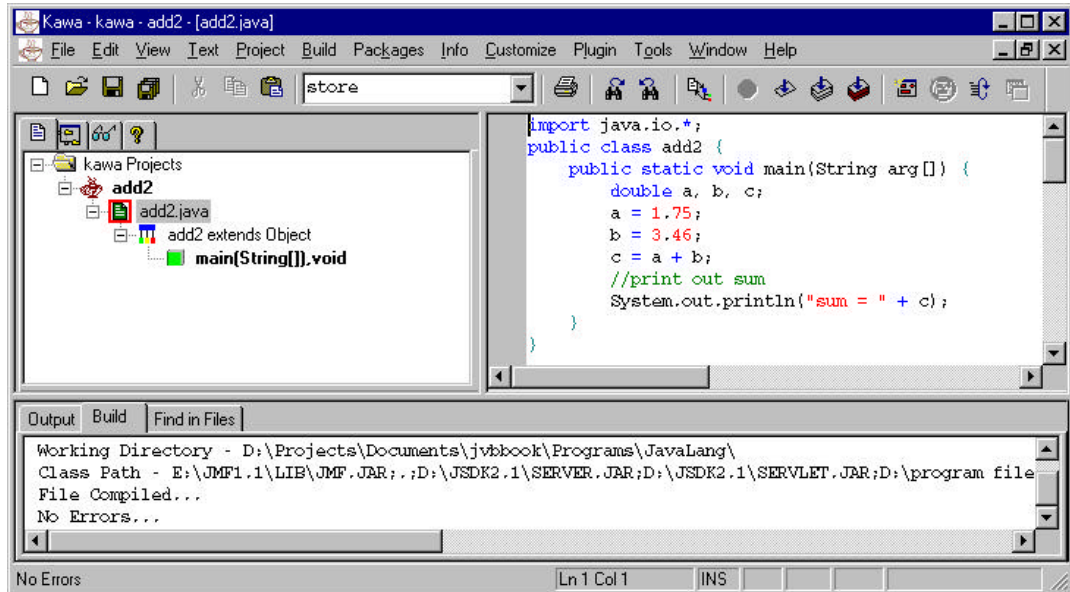
Once you download and install Kawa, you need to tell it where to find your Java development kit executables, usually in `\jdk1.x\bin`. Then you can create a project and add files to it using Project | New. In Figure 6-1, we see the Kawa editor with the simple Add2 program displayed.



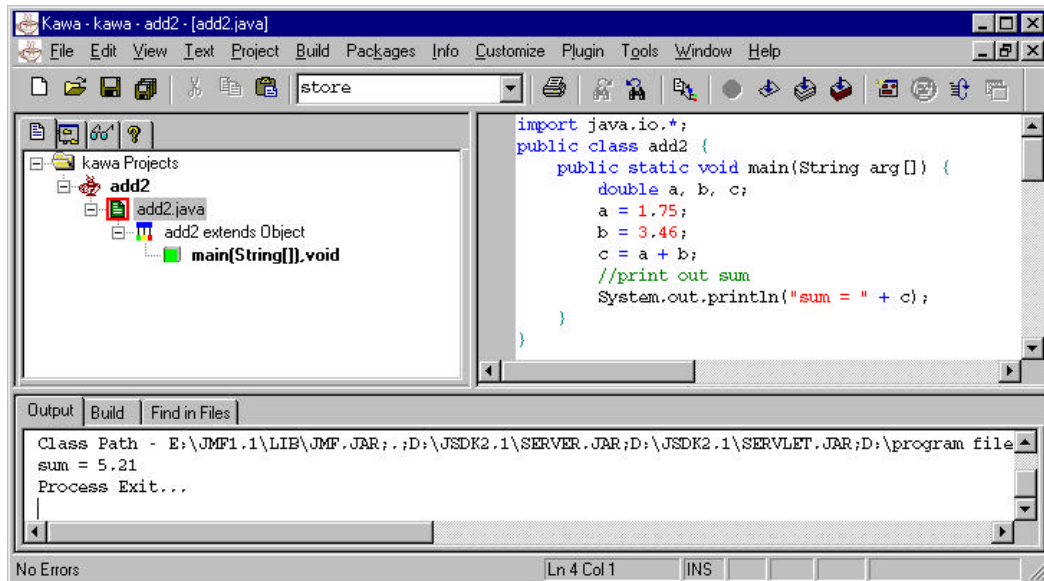
Note that the program files are on the left, and the current file selected displayed on the right. Once you have typed in a program, you can compile it by selecting Build | Compile, or just pressing F7. To see the results of the compilation, select View | Output and drag the lower window splitter bar down to show the output window.



If there are any compilation errors, you can double click on them to move your editing cursor to the offending line in the source code window. When you have corrected the errors, press F7 to recompile.



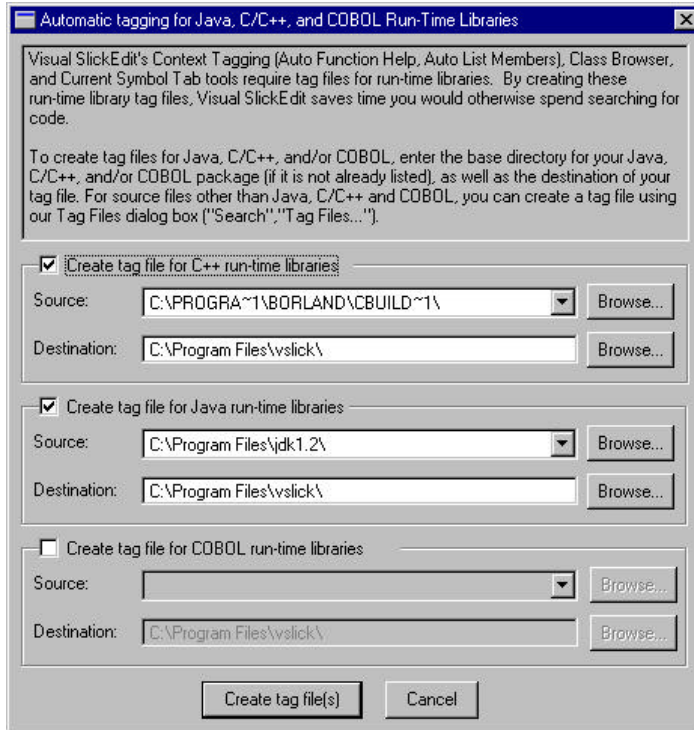
Then, to execute the program, Select Build | Run or just press F4.



The program output is shown in the output window.

Using Visual SlickEdit

Ideally, you should install the Java Development Kit (JDK) and source code first and then install Visual SlickEdit. If you do, then it will ask you for the path to the Java libraries.



You can set these up later by Selecting Search | Tag files | Add tag files. Once you have set up these tag files for Java, Visual SlickEdit will automatically suggest syntax to complete a given Java statement.

For example, when we type in “System.out.” VSlick will display the Java methods we might select to complete the statement.

```

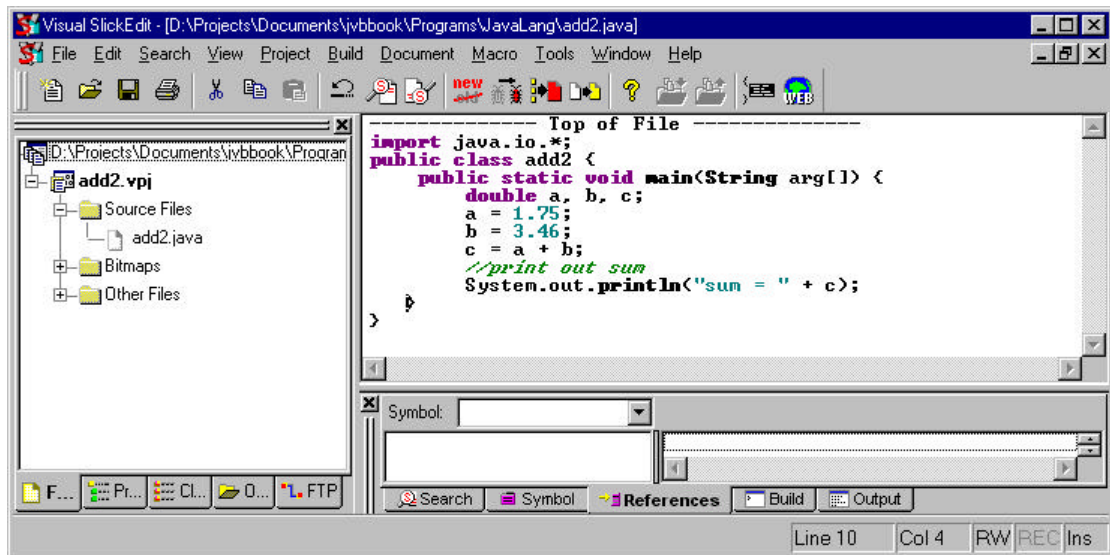
----- Top of File -----
import java.io.*;
public class add2 {
    public static void main(String arg[]) {
        double a, b, c;
        a = 1.75;
        b = 3.46;
        c = a + b;
        //print out sum
        System.out.println("sum = " + c);
        System.out.
    }
}

```



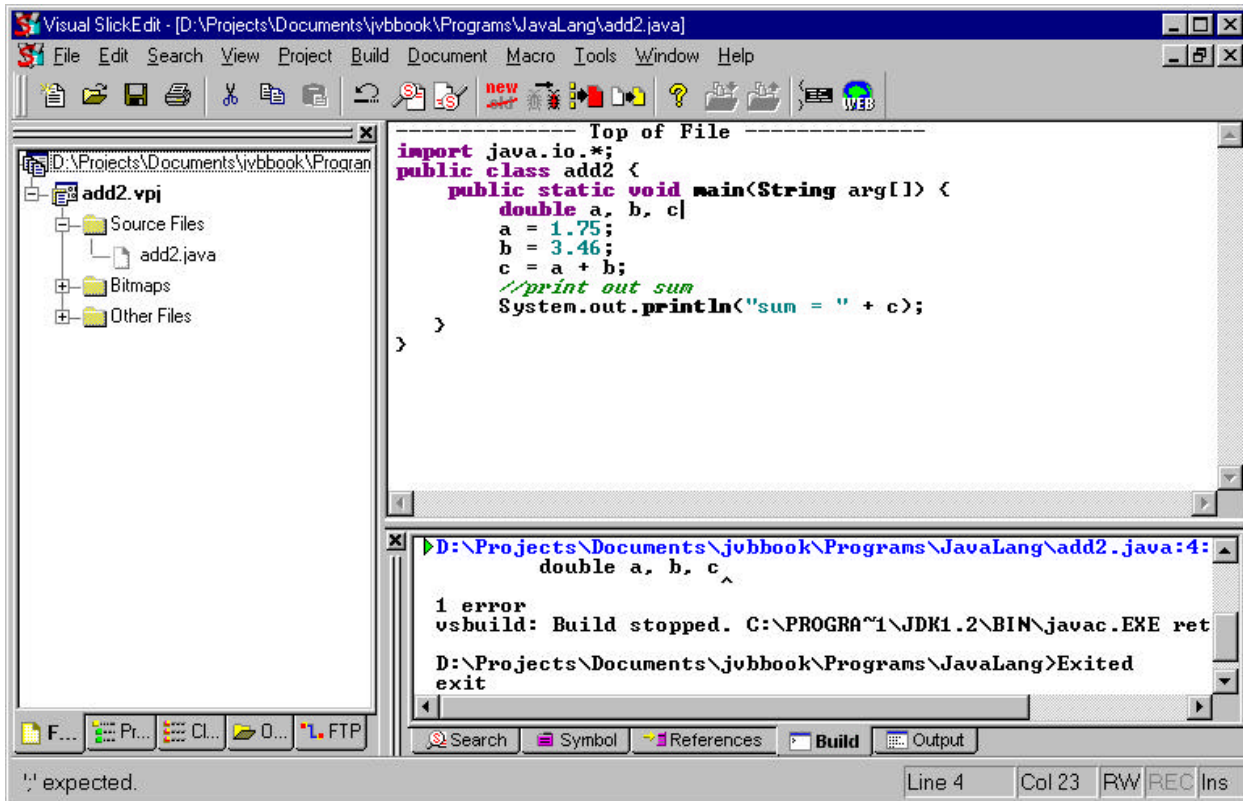
To create a project in Visual SlickEdit, select Project|New and select a folder to store the files in. It is best if all the files in a project are in the same directory or subdirectories beneath it.

We show the add2.java program project below.



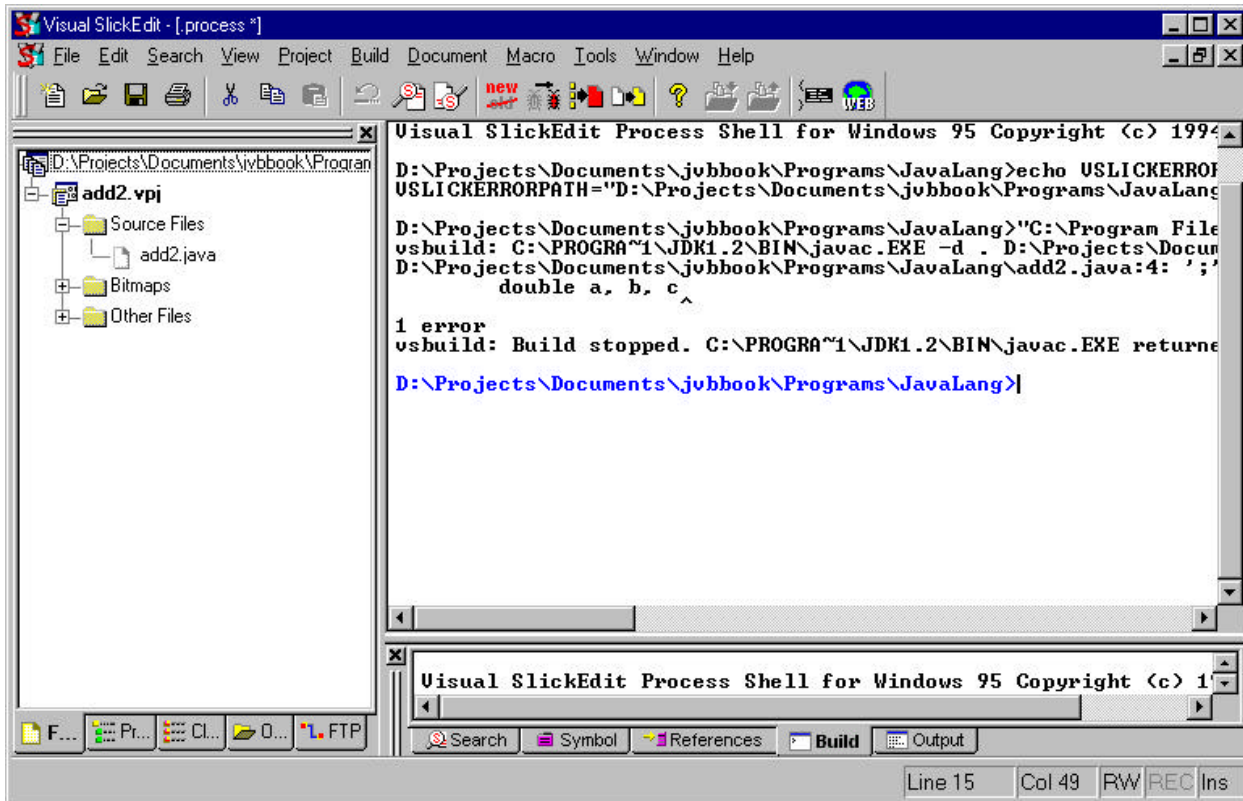
Note that the project consists of source files as well as Bitmaps and other files, such as HTML or data files that you want to make sure you keep with the project.

To compile the project, select Build | Build or press Ctrl/M. Error messages are displayed in the Build tab of the lower window.



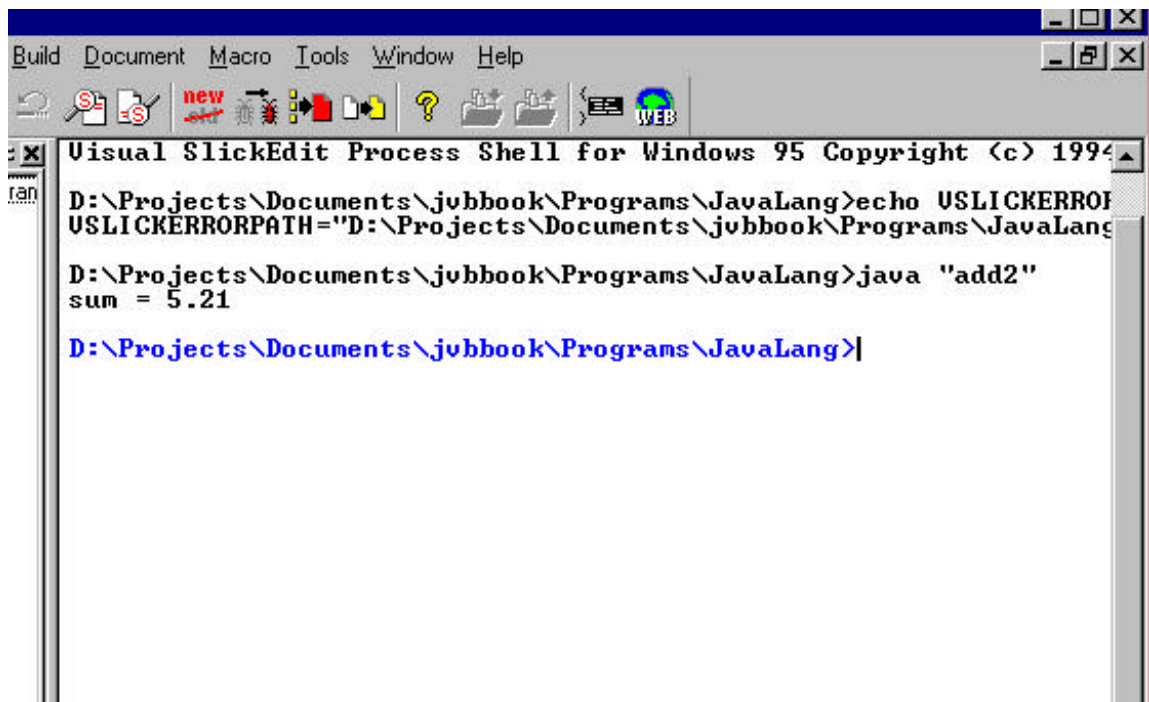
and you can double click on the top line of each error message to have the cursor placed on the error line.

You can also choose to have the error messages displayed in the main window, by selecting Project | Project properties and selecting “Capture Output to Process Buffer.”



You can still double click on any error message to have the cursor move to the offending line.

To execute the program, once you have removed all the errors, select Build|Execute or Alt/B X. The output will show in the output window.



```
Visual SlickEdit Process Shell for Windows 95 Copyright (c) 1994
D:\Projects\Documents\jobbook\Programs\JavaLang>echo USLICKERROR
USLICKERRORPATH="D:\Projects\Documents\jobbook\Programs\JavaLang
D:\Projects\Documents\jobbook\Programs\JavaLang>java "add2"
sum = 5.21
D:\Projects\Documents\jobbook\Programs\JavaLang>|
```

Summary

We've seen a brief outline of how you can use both Visual SlickEdit and Kawa for Java development in this chapter. They each have a number of advantages and we recommend both of them for beginning programmer

6. Object Oriented Programming

There are two kinds of people: those who divide everything into two categories and those who don't. Likewise there are two kinds of programming: procedural and object oriented. In this chapter we'll begin to understand what objects are and why they make programming easier and less prone to errors.

A *procedural* program is written in the style you are probably most familiar with: one in which there are arithmetic and logical statements, variables, functions and subroutines. Data are declared somewhere at the top of a module or a procedure and more data are passed in and out of various functions and procedures using argument lists.

This style of programming has been successfully utilized for a very long time as programming goes but it does have some drawbacks. For example, the data must be passed correctly between procedures, making sure that it is of the correct size and type, and the procedures and their calling arguments may often need to be revised as new function is added to the program during development.

Object-oriented programming differs in that a group of procedures are grouped around a set of related data to construct an *object*. An object is thus a collection of data and the subroutines or *methods* that operate on it. Objects are usually designed to mimic actual physical entities that the program deals with: customers, orders, accounts, graphical widgets, etc.

More to the point, most of *how* the data are manipulated inside an object is invisible to the user and only of concern inside the object. You may be able to put data inside an object and you may be able to ask to perform computations, but how it performs them and on exactly what internal data representation is invisible to you as you create and use that object. Once someone creates a complete, working object, it is less likely that programmers will modify it. Instead they will simply derive new objects based on it. We'll be taking up the concept of deriving new objects in Chapter 6.

Objects are really a lot like C structures, Pascal records or Visual Basic types except that they hold both functions and data. However, objects are

just the structures. In order to use them in programs, we have to create variables of that object type. We call these variables *instances* of the object.

A Procedural Program

Let's take a very simple example. Suppose that we want to draw a rectangle on our screen. We need to design code for drawing a rectangle and specifying what size it is (as well as where). Now, our first thought might have been to simply write a little subroutine to draw the rectangle, and then draw each rectangle we want by calling this subroutine:

```
Call DrawRect (10, 10, 200, 150)
Call DrawRect (40, 40, 150, 100)
```

The problem with this program is that the drawing routine has to know a lot about the display characteristics. We call the routine using pixel coordinates, but we might have to convert the coordinates to another coordinate system. This DrawRect routine is publicly available for modification throughout the program, and if we need more arguments, such as color or line thickness, we would just have to add them to an ever growing list of subroutine arguments. Further, if we wanted to draw some other shapes, we'd have to go through the same coordinate conversion for that routine as well, in addition to adding arguments to it if we decided we needed to specify color or line thickness.

An Object-Oriented Program

If we were to rewrite this program using object-oriented style, we would design a rectangle object which could *draw itself*. We wouldn't have to know how it did the drawing or whether the algorithm or calling parameters changed from time to time.

We do this by creating a rectangle **class** with a **draw** method inside it. Let's look at how we would do this in Java. The procedure is quite analogous:

- We create a rectangle **class**
- We create *instances* of that class, each with different sizes

- We ask each instance to draw itself.

A compiled Java object is called a **class**. Remember that in our very first simple program in chapter 3, we used the key word "class" in creating the outer wrapper of our example program. Each Java class is an object which can have as many instances as you like.

When you write a Java program, the entire program is one or more classes. Usually, each class is in a separate source file. The main class represents the running program itself, and it must have the same name as the program file. In the rectangle example, the program is called Rect1.java and the main class is called Rect1.

Classes in Java contain data and functions, which are called *methods*. Both the data and the methods can have either a **public** or a **private** modifier, which determines whether program code outside the class can access them. Usually we make all data values **private** and write public methods called *accessor methods* to store data and retrieve it from the class. This keeps programs from changing these internal data values accidentally by referring to them directly.

If we want users of the class to be able to use a method, we, of course, must make it public. If on the other hand, we have functions which are only used inside the class, we would make them private.

While a Java program can be made up of any number of .java files, each file can contain only one public class and it must have the same name as the file itself. There may be any number of additional classes within the file which are not declared as public, but by convention, programmers usually put each class in a separate file.

Creating Instances of Objects

We use the **new** operator in Java to create an instance of a class. For example to create an instance of the Button control class, we could write:

```
Button Draw; //a Button object  
  
//create button with "Draw" caption
```

```
Draw = new Button("Draw");
```

Remembering that we can also declare a variable just as we need it, we could also write somewhat more compactly:

```
Button Draw = new Button("Draw");
```

Don't be confused by the fact that the variable name and the caption of the button are the same: they are unrelated, but convenient in many programs.

Remember, while we can create new variables of the primitive types (such as int, float, etc.) we must use the **new** operator to create instances of objects. The reason for this distinction is that objects take up some block of memory. In order to reserve that memory, we have to create an instance of the object, using the **new** operator.

Constructors

When we create an instance of a class we write ourselves, we usually need to write code that initializes variables inside the object. This code is put in the class's *constructor* routine. A constructor routine has the same name as the class, is always public, and has no return type (not even **void**).

```
public class Rectangl {
    private int xpos, ypos;
    private int width, height;

    public Rectangl(int x, int y, int w, int h) {
        xpos = x;        //remember size and posn
        ypos = y;
        width = w;
        height = h;
    }
}
```

In the above example, our Rectangl class saves the rectangle position and size. We could also save the line color.

A Java Rectangle Drawing Program

In the example below, we see a complete Rectangl class, including its **draw** routine. The drawing takes place using the windows Graphics object.

```

import java.awt.*;

public class Rectangl {
    private int xpos, ypos;
    private int width, height;

    public Rectangl(int x, int y, int w, int h) {
        xpos = x;        //remember size and posn
        ypos = y;
        width = w;
        height = h;
    }
    //-----
    public void draw(Graphics g) {
        //draws rectangle at current position
        g.setColor(Color.blue);
        g.drawRect(xpos, ypos, width,height);
    }
}

```

We spell Rectangl without the “e” because Java already has a class named “Rectangle.”

The Main Rect1 Program

The main program creates two *instances* of the Rectangl class with different size and shape. In a windowing environment, the screen needs to be repainted whenever windows are resized or moved, and when this occurs the Java system calls this class’s paint method and passes it the Graphics object it needs to do the drawing. We pass this Graphics object on to the two instances of the Rectangl class, and each one draws a rectangle where that one is supposed to appear.

```

//Simple Rectangle drawing example

public class XRect1 extends XFrame {
    private Rectangl rect1;        //two rectangle objects
    private Rectangl rect2;
    //-----
    public XRect1() {
        super("XRect1 window"); //window title bar
        //Create rectangles and tell them where to draw
        rect1 = new Rectangl(10, 40, 200, 100);
        rect2 = new Rectangl(70, 50, 150, 75);

        setBounds(50, 50, 475, 225); //size of window
        setVisible(true);           //display window
    }
}

```

```

    }
//-----
    public void paint(Graphics g) {
        rect1.draw(g);
        rect2.draw(g);
    }
//-----
    public static void main(String args[]) {
        new XRect1();
    }
}

```

This is a complete working program as shown and is called XRect1.java on the example disk in the \chapter4 directory. The XRect1 class is derived from (and uses) the XFrame class. We'll explain how the Xframe class works in a later chapter, but we will just note here that by using XFrame, you can write programs whose windows will close when you click on the close box in the upper right corner, while if you do not use the Xframe class, you may not be able to close your window easily.

Let's look at how the program works. You can see its window display in Figure 4-3.

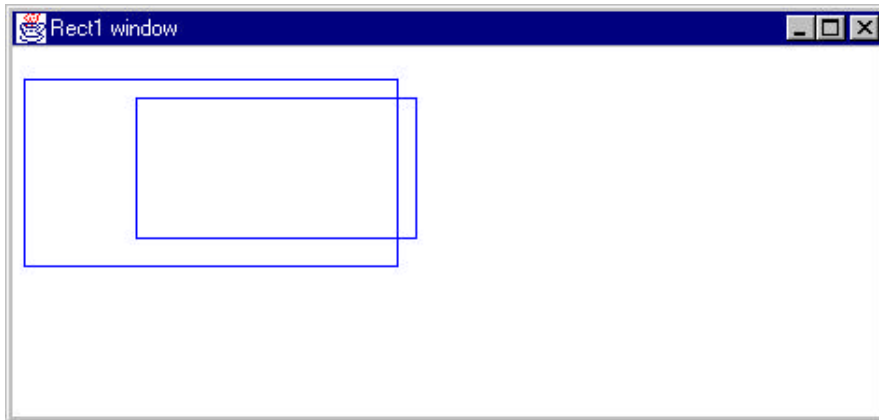


Figure 4-3: The XRect1 program for drawing two rectangle objects.

First, the **main** routine is where the program actually starts. If you want to write a stand-alone application one and only one of its classes must have a main routine, and its "signature" must be *exactly*:

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

While that main routine appears to be part of a class, it is actually in a way grafted on (because of the **static** qualifier) and it is the entry point for the program. All this main routine does is create one instance of the program class XRect1.

```
new XRect1();
```

Constructors

When you create an instance of an object, the code in the *constructor* routine is executed automatically. This is the place where we will initialize the window variables and create instances of the rectangle object.

```
public XRect1()          //window class constructor
{
```

The first line initializes the window itself, creating a title bar title by passing it to the parent Frame class.

```
    super("XRect1 window"); //create window with title
```

Then, we create two instance of the Rectangl class and define their shapes.

```
    //Create rectangle and tell them where to draw
    rect1 = new Rectangl(10, 40, 200, 100);
    rect2 = new Rectangl(70, 50, 150, 75);
```

Now we have two rectangle objects we can draw whenever we need to by simply telling each one to "draw itself."

```
    rect1.draw(g);
    rect2.draw(g);
```

We do just this inside the paint method.

Methods inside Objects

As we noted above, functions inside a class are referred to as *methods*. These functions can be **public**, meaning that you can access them from

outside the class, **private**, meaning that they can only be accessed from inside the class and **protected**, meaning that the methods can be accessed only by other classes in the same file.

Syntax of Methods

A method inside an object is just a function or subroutine. If it returns a value, you declare the type of the return

```
int getSize() {
}
```

If it returns no value, like a subroutine in other languages, you declare that it is of type **void**.

```
void setSize(int width, int height) {
}
```

In either case you must declare the type of each of the arguments. It is usual to use descriptive names for each of these arguments so the casual reader can figure out what each method does.

Variables

In object oriented programming, you usually make all of the variables in a class **private** as we did above with **xside** and **yside**. Then you set the values of these variables either as part of the constructor or using additional **set** and **get** functions. This protects these variables from accidental access from outside the class and allows you to add data integrity checks in the **set** functions to make sure that the data are valid.

We could, of course, have made the `Rectangle`'s **height** and **width** variables **public** and set them directly.

```
rect1.width = 300;
rect1.height = 200;
```

but this gives the class no protection from erroneous data such as:

```
rect1.width = -50;
```

So instead, we use *accessor* functions such as **setSize** to make sure that the data values we send the class are valid:

```
rect1.setSize(300, 200); //call resize method
```

and then within the class we write this accessor function with some error checking:

```
public void setSize(int w, int h) {
    if (w > 0 )
        width = w; //copy into width if legal
    if (h > 0)
        height = h; //and into height if legal
}
```

Capitalization Conventions in Java

While you can choose to capitalize any way you want to, most Java developers choose to capitalize as follows.

1. Class names usually start with a capital letter:

```
public class Rectangl
```

2. Instances of class usually begin with a lower case letter:

```
Rectangl rect = new Rectangl();
```

3. Methods inside classes usually begin with a lower case letter but use capital letters to set off the other word boundaries in the name. In particular, get and set methods always start in lower case.

```
public void setWidth();
public int getWidth();
```

4. Constants are usually all upper case, with the words separated by underscores:

```
final int KEY_DOWN = 5;
```

Multiple Methods with the Same Name

Our Rectangl class had one constructor, containing the dimensions of the shape.. However, we can have more than one constructor, and in fact more than one version of any method, *as long as they have different argument lists or signatures*. So we could declare our rectangle's shape at the same time we create it:

```
public void Rectangl(int width, int height) {
    width = w; //save width and height
    height = h;
}
```

The Java compiler must be able to distinguish these various versions of the same method by either the number or the type of arguments. If two different methods have the same number and type of arguments, the compiler will issue an error message.

Passing Arguments by Value

All variables are passed into methods by *value*. In other words their values are *copied* into new locations which are then passed to the subroutine. So, if you change the value of some argument to a method, it will not be changed in the original calling program.

For example, suppose we need to swap the values of two integers several times in some program. We might be tempted to write a private swap method like this:

```
private void swap (int x, int y) {
//this looks like it will swap integers,
//but it won't
    int temp = x;
    x = y;
    y = temp;
}
```

and then call this method from the main class:

```
int a = 5; //assign two values
int b = 10;
swap (a, b); //integers passed by value
System.out.println("a=" + a + " b=" + b);
```

However, we find that **a** still is 5 and **b** still is 10. And of course the reason why this won't work is that the memory locations containing the arguments *x* and *y* are *copies* of the original calling parameters, and switching them will not switch the originals.

Objects and References

Objects, on the other hand, are always referred to by reference. Since objects take up a number of bytes of memory, this reference is actually a *pointer* to that block of memory. While actual pointers to memory locations don't exist at the programmer level in Java, references are implemented under the covers as pointers. Now, when an object is passed into a method as an argument, Java passes in a copy of that reference. So, if you manipulate the values contained in an object inside some other method, you do indeed change the values in the original object you passed into the routine.

Now as we noted earlier, there are object versions of the primitive types, called Integer, Long, Float, and Boolean. We might at first imagine that we could write a method which would swap objects since they are passed into the method by reference.

```
private void Swap(Integer x, Integer y) {
//swaps Integer objects, but to no avail
  Integer temp = new Integer(x.intValue());
  x = y;
  y = temp;
}
```

However, if we write a program to use this method, we find, sadly, that this method still does not affect the calling parameters.

```
Integer A = new Integer(a);
Integer B = new Integer(b);
Swap (A, B);           //objects passed by reference
System.out.println("a=" + A + " b=" + B);
```

Why is this? What's going on here? Well, remember that the values passed into the Integer class Swap method are pointers to the original objects A and B. Let's suppose that they have the address values 100, and 110. Then when we create the new Integer **temp** let's suppose its address is 300. After the Swap method completes, x will have the value 110 and y will have the value 300. In other words, we changed the values of the pointers, but did not change the contents of the objects. In addition, these are just local copies of the pointers to these objects, so we haven't swapped the references in the calling program in any case. In fact, the Integer class does not have any methods to change its contents, only to create new Integers, which will not help us in this case.

So even though objects are passed into methods by reference, changing the values of the pointers we used does not change the contents of the objects themselves. If we want to make an effective swap, we would have to create an object whose *contents* we could change.

For example, we could create a `Pair` class with a `swap` method:

```
public class Pair {
    private Object a;          //two objects we will swap
    private Object b;
    //-----
    public Pair(Object x, Object y) {
        a = x;
        b = y;
    }
    //-----
    public Pair(int x, int y) {
        a = new Integer(x);
        b = new Integer(y);
    }
    //-----
    public void swap() {
        Object temp = a;    //swaps values of objects
        a = b;
        b = temp;
    }
    //-----
    Object geta() {
        return a;    //returns values of objects
    }
    //-----
    Object getb() {
        return b;
    }
}    //end Pair class
```

and accessor methods **geta** and **getb** to obtain the values after swapping. Then we could call these methods as follows:

```
Pair p = new Pair(A, B); //copy into object
p.swap();                //swap values inside object
A =(Integer)p.geta();   //get values out
B =(Integer)p.getb();
System.out.println("a=" + A + " b=" + B);
```

And, if you think this is silly in this case, you are right. After all, swapping the values of two numbers or objects only requires 3 lines of code in any

case. However, this does illustrate an important concept. If you want to change values, you have to do it inside the object, not by switching copies of the references to the objects.

Object Oriented Jargon

Object-oriented programs are often said to have three major properties:

- **Encapsulation** - we hide as much as possible of what is going on inside methods in the object.
- **Polymorphism** - Many different objects might have methods having identical names, such as our draw method. While they may do the same thing, the way each is implemented can vary widely. In addition, there can be several methods within a single object with the same name but different sets of arguments.
- **Inheritance** - objects can inherit properties and methods from other objects, allowing you to build up complex programs from simple base objects. We'll see more of this in the chapters that follow.

Summary

First let's review some terminology:

1. Objects in Java are created using classes.
2. Each class may have one or more constructors, none of which have a return type.
3. Functions inside the class are called *methods* and may be public or private.
4. Each variable whose type is declared to be of that class is called an *instance* of that class.
5. Variables inside the class are usually private and are referred to as *instance* data since each instance of the class may have different values for these variables.

6. One and only one class per Java application may have a public static method called *main*, where the program actually begins.

7. Using Classes in Java Programming

Now that we've seen how simple creating objects is in Java, it won't surprise you to discover that *everything* in Java is accomplished using classes. There are no library functions or independent subroutines in Java: only objects of various types and their methods. While this may take a slight attitude readjustment, you'll quickly see that the consistency this approach brings to Java makes it a very easy language to learn and use.

The String Class

Strings in Java are one of the most commonly used objects and contain a fairly rich set of methods for manipulating character strings. Strings are not arrays, but you can manipulate groups of characters in an analogous manner. Remember that Strings contain 16-bit Unicode characters, so that they can represent a wide variety of fonts and languages.

String Constructors

The fact that an object may have any number of constructors, each with different arguments is another example of polymorphism. The most common string constructor is

```
String s = new String("abc");
```

but you can also create a string from an array of characters from some file or network socket:

```
String(char[])                //from array of char  
  
//from specified part of array of char  
String(char[], int offset, int count)  
  
//from array, setting upper byte  
String(char[], int hibyte, int offset, int count)
```

The third constructor above is used to assure that the upper byte of each 16-bit character is set to a known value, usually zero. This function is very important in Windows 95, where Java 1.0 otherwise sets this upper byte to an indeterminate value, making comparisons with other strings impossible.

String Methods

There are a wide variety of methods in the String class. Some of the most common are:

```
length()
equals(String)
startsWith(String)
endsWith(String)
toUpperCase()
toLowerCase()
indexOf(String)
substring(int begin)
substring(int begin, int end)
```

To reiterate, these are *methods* that operate on a String object, not functions to be called with a string as argument. Thus, to obtain the length of a string, you might perform the following steps:

```
//create an 8-character string
String abc = new String("alphabet");
int len = abc.length();           //len now contains 8
```

You can look over the plethora of other string methods in the String documentation provided with the SDK.

The String +-operator

The +-sign in Java is said to be “overloaded” with respect to strings. Thus, you can combine strings much as you can in VB and Pascal:

```
String h = new String("Holiday");
String fs = new String("for Strings");
```

```
String title = h+ " " + fs;
//prints "Holiday for Strings"
System.out.println(title);
```

You can also use the `+` operator to combine basic numeric types with strings. They are automatically converted to strings:

```
int count = 24;
System.out.println("Found " + count + " blackbirds");
// prints out "Found 24 blackbirds"
```

Note that there are no leading or trailing spaces in numbers produced in this fashion and you must be sure to include them in your code.

Conversion of Numbers to Strings and Vice-versa

You can convert any simple numeric type (int, float, double, etc.) to a string in one of two ways:

The simplest way is to convert using the `String` class's `valueOf()` method.

```
int length = 120;
String strLength = new String().valueOf(length); //returns a
string "120"
```

There is a version of this method for each of the basic types: in other words this method shows polymorphism.

You can accomplish the same thing using the `toString()` methods of the **Integer**, **Float** and **Double** classes, which are object classes wrapped around the base numeric types:

```
int length = 120;
String strLength = new Integer(length).toString();
```

To convert a `String` to a number, you can use the `intValue()`, `floatValue()` and related versions of the `Integer` and `Float` classes:

```
String strLength = new String("120");
int length = new Integer(strLength).intValue();
```

Changing String Contents

The `String` class is designed to be *immutable*: once you have created a string you cannot change its contents. The `StringBuffer` class is provided so you can change individual characters of a `String` and then put the changed result back into a string.

You can create an instance of the `StringBuffer` class from a string:

```
String alph = new String("abcde");
StringBuffer buf = new StringBuffer(alph);
```

Then you can examine or change any character using the following methods:

```
public char charAt(int n); //get char at posn n
public void setCharAt(int n, char ch); //set char
public StringBuffer insert(int n, char c);
```

as well as with a host of other useful methods listed in the documentation. When you have changed the characters in the string buffer, you can regenerate the string with the `toString` method.

```
alph = buf.toString();
```

The StringTokenizer

It is often useful to break a set of text apart into tokens, where the characters separating the tokens can be specified. The `StringTokenizer` class does that. You create an instance of the `StringTokenizer` using a `String` and an optional second string of separator characters. Then you can use the `hasMoreTokens` and `nextToken` methods to obtain the tokens in the string.

```
String fString="Apple|Orange|Pear|Sour Grapes|";
```



```
StringTokenizer token =
    new StringTokenizer(fString, "|");
while(token.hasMoreTokens()) {
    System.out.println(token.nextToken());
}
```

The Array Class

Arrays are a built-in class whose syntax is part of the language, much as Strings are. Arrays can be singly or multiply dimensioned and may consist of any base numeric type or of any object. You declare an array object by

```
float x[] = new float[100]; //dimension array
```

and you can access it by enclosing the index in brackets. Note that array indices always begin at 0 and end at one less than the array dimension:

```
for (i=0; i<100; i++)
    x[i] = i;
```

When you declare a new array, its elements are initialized to 0 if it is numeric or to **null** if it is an array of objects. You can also declare specific contents for an array:

```
int a[] = new int[5];
a[] = {1, 3, 5, 7, 9};
```

You can also declare arrays of more than one dimension by including several dimensions in successive brackets:

```
float x[][] = new float[12][10];
```

```
int z[][][] = new int[3][2][3];
```

Because Java actually handles these multidimensional arrays as arrays of *objects*, each with their own dimensions, you do not have to specify all of the dimensions in the initial declaration. The leftmost dimensions must be specified but dimensions to the right may be omitted and defined later:

```
float abc[][] = new float[100][];
```

Here, **abc** is actually a single-dimensional array of float[], where these dimensions are not yet defined.

Garbage Collection

Once we begin allocating large amounts of memory in any program, we are naturally concerned about how that memory can be released when we are done with it. In fact, some of the most annoying bugs in programs in other languages come from allocating but not releasing memory correctly.

In Java, this is never your concern, because the Java run time system automatically detects when objects are no longer in use and deletes them. Thus, while we can use the **new** command prolifically to allocate memory as we need it, we never have to concern ourselves with the *management* of that memory and its subsequent release.

Vectors

A Vector is an unbounded array. You can put any kind of objects into a vector using the addElement method and fetch them out again using the elementAt(*i*) method.

```
Vector fruit = new Vector();
fruit.addElement("Apple");
fruit.addElement("Peach");
fruit.addElement("Watermelon");
```

```
String myFruit = (String)fruit.elementAt(1);
```

Vectors are ideal when you don't know in advance what size your array might become. The only disadvantage to Vectors is that they return elements of type Object, and you have to cast them to the correct type, as we show above.

Vectors have a `size()`, and the elements are numbered from 0 to `size() - 1`.

```
for(int i=0; i < fruit.size(); i++)
    System.out.println(fruit.elementAt(i));
```

Hashtables and Hashmaps

A Hashtable is like a Vector, except that each object in the array has a unique *key*. The key can be a number or any other object. While the values in the Hashtable can be duplicates, the keys must be unique. Hashtables provide a convenient way of maintaining a list of unique items, where you can use the key to fetch an item or find out if one having that key exists already. Some of the more useful methods are:

```
public Object put(Object key, Object value);
public Object get(Object key);
public boolean containsKey(Object key);
public int size();
```

The HashMap, added in Java 2, provides much the same interface, but allows both value and key objects to be null. Unlike the Hashtable, it is not synchronized, so it is possible for two threads to modify it at once, leading to possible errors.

Constants in Java

If you have been programming in VB, C++ or Pascal, you are probably familiar with named constants. These are used whenever you can improve program readability. Named constants do not change during a program's execution, but you may elect to change their values during the development of a program.

```
//In C or C++
const PI = 3.1416;
```

In Java, values which cannot be changed are said to be **final**, and as usual must be members of some class. If you make reference to such constants, you must refer to the class they are members of as well. For example, the **Math** class contains definitions for both *pi* and *e*.

```
float circumference = 2 * Math.PI * radius;
```

Similarly, most of the common colors are defined as RGB constants in the **Color** class:

```
setBackground(Color.blue);
```

You can declare constants by making them **final** and **public** if you want to access them from outside the classes.

```
class House
{
public final int GARAGE_DOORS = 2;
...
}
```

Then, whether or not you have a current instance of the House class, you can always refer to this constant:

```
System.out.println("Doors =" + House.GARAGE_DOORS);
```

Note also that Java has three “built-in” constants” **true**, **false**, and **null**.

Summary

In this chapter, we’ve learned about the built-in Java String, StringBuffer and Array classes and how to print out numbers as strings. We’ve also touched on Java’s automatic garbage collection and on how to use named constants in classes. Now, we’re ready to take up inheritance as the last major new topic before we complete our first tour around the Java language.

8. Inheritance

The greatest power of programming in object oriented languages comes from *inheritance*: the ability to make new, more versatile objects from already completed objects, without changing the original objects. One of the reasons this approach is so powerful in actual code development is that it allows you to write new classes based on existing classes without changing the existing class in any way. Thus, if you have working code in one class, you don't risk "breaking" it by writing modifications..

To see how we can use inheritance in Java, lets consider a simple applet called **Rect2**, an applet analog of the application we wrote in chapter 4. Remember that applets can run only on web pages and have extremely limited access to your computer's resources. An applet does not have a **main** routine but starts directly in the constructor of the class you have declared **public**.

```
//This program draws a rectangle
public class Rect2 extends XFrame {
    Rectangl r1; //one instance of the rectangle object

    public Rect2()      { //constructor
        super("Rectangle");
        r1 = new Rectangl(10,50,150, 100);           //create
rectangle
        setBounds(100,100,300,300);
        setVisible(true);
    }
//-----
    public void paint(Graphics g) {
        r1.draw(g);
    }
//-----
    static public void main(String argv[]) {
        new Rect2();
    }
}
```

and the accompanying Rectangl class

```
public class Rectangl {
    private int width, height;
```

```

private int xpos, ypos;
private Color color;

public Rectangl(int w, int h) {
    // sets w & h
    width = w;
    height = h;
}
//-----
public void move(int x, int y) {
    xpos = x;
    ypos = y;
}
//-----
public void setColor(Color c) {
    color = c;
}
//-----
public void draw(Graphics g) {
    g.drawRect(xpos,ypos, width, height);
}
}

```

This simple program consists of two classes, **Rect2** and **Rectangl**. The **Rect2** class is the **public** class and is the one launched by the applet. To run this applet, you compile it as usual by

```
javac Rect2.java
```

and then execute it by embedding it in a simple web page, rect2.html.

```

<html>
<Head><title>Draw Rectangle Applet</title></head>
<body>
<applet code="Rect2.class" height=200 width=300>
</applet>
</body>
</html>

```

You will note that the **Rect2** class is related to the XFrame class

```
public class Rect2 extends Xframe
```

which we have written to provide a frame that exists easily when you click on the close box.

The **extends** keyword indicates that the **Rect2** class is derived from or inherits from the **XFrame** class. Thus, this new **Rect2** class is not just an *instance* of XFrame but a class which has all the properties of XFrame *in*

addition to any properties you might change. For example, in this simple class, there are only two methods: the constructor and the **paint** method. In the constructor method, we create an object of type **Rectangl** of size 150 x 100. (As before, we avoid the spelling “Rectangle” with an “e” because the Java system already has such a class).

The **paint** method is called by the underlying window system of your computer operating system through the Java runtime system whenever some action occurs that requires that the window be repainted. In this case, we replace whatever the system might do during a paint method with a call to the **draw** method of the rectangle we just created in the constructor.

Our **Rectangl** class also consists of only two methods, a constructor, which saves the dimensions inside the object, and a **draw** method which actually draws the rectangle on the screen. Fortunately the Java graphics system has a **drawRect** method so we don’t have to draw it a line at a time. The displayed window is shown in Figure 6-1.

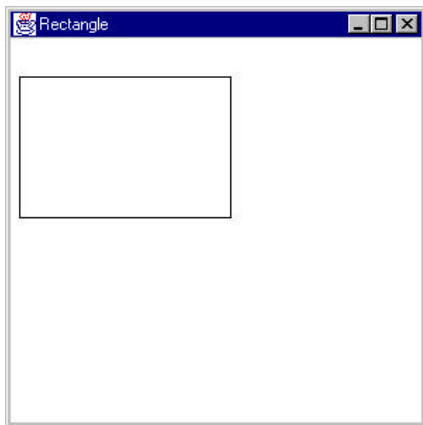


Figure 8-1The Rect2 program

Drawing a Square

Now a square is just a special case of a rectangle, so it is not surprising that we can derive a **Square** class from the **Rectangl** class. We will declare this class as follows:

```
public class Square extends Rectangl {
```

```
//This square class is derived from Rectangl
public Square(int side) {
    super(side, side); //sides are the same size
}
}
```

The **super** method call in the **Square** constructor means to pass the arguments to the parent class: in this case the **Rectangl** class. In other words, it calls the **Rectangl** constructor with both sides equal. This **super** method must always be the *first* line of code in the constructor. It passes data up the inheritance chain where it initializes all the parent classes.

Now you might ask, where's the square? How do we draw it? Well, we don't. The **Square** class has no methods of its own. It *inherits* all the methods of the **Rectangl** class and thus, the **draw** method is contained automatically. We simply pass the height and width values into the **Rectangl** class and use all of its methods. The complete program shown below is identical to the **Rect2** program except for the small, new **Square** class.

```
public class Sqr extends XFrame {
    Square s1; //one instance of the square

    public Sqr()    { //constructor
        super("Draw square");
        setBounds(100, 150, 150, 150);
        s1 = new Square(20, 40, 100); //create square
        setVisible(true);
    }
    //-----
    public void paint(Graphics g) {
        s1.draw(g);
    }
    static public void main(String argv[]) {
        new Sqr();
    }
}
```

The simple Square class is derived from Rectangle and is merely

```
public class Square extends Rectangl {

    public Square(int xp, int yp, int side) {
        //sides are the same size
        super(xp, yp, side, side);
    }
}
```


}
 Again, note that the **Square** object has no specific **draw** method, but that it *inherits* one from its parent **Rectangl** class. It also inherits any methods that the parent of the parent contain as well, as far back as the inheritance tree extends. The only requirement is that these methods be declared **public** in the base class, and if they are overridden in descendant classes that these be declared **public** as well.

Figure 6-2 shows this **Sqr** application running, and the files `Sqr.java` and `Sqr.html` in the `\chapter6` directory of the example disk contain the code.

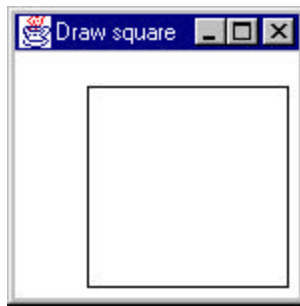


Figure 8-2 The Sqr program, showing the Square class derived from the Rectangl class.

Inheritance Terminology

There are a number of interchangeable terms used regarding inheritance. We can say that the square class is *derived* from the rectangle class, or that it is a *subclass* of the rectangle class. We also can say that the square class *extends* the rectangle class. Finally, we might say that the square class *inherits* from the rectangle *super class*. If you write a method in the derived class which has the same name and calling arguments as one in the parent class, you have *overridden* that method in the parent class.

The instanceof Operator

Java provides the **instanceof** operator that you can use to discover whether an object is derived from a given class:

```
Square sq1 = new Square(50);
```

```
if (sql instanceof Rectangl)
    System.out.println("Is a rectangle or descendant");
```

Overriding Methods in Your Derived Class

Now lets suppose that we want to always make sure that squares are drawn in red and with lines twice as thick, but that rectangles can be drawn in any color. First, we'll add a color parameter to the rectangle class and a **setColor** method to store colors with each rectangle object. This new class looks like this:

```
public class Rectangl {
    private int width, height;
    private int xpos, ypos;
    private Color color;

    public Rectangl(int w, int h) {
        width = w;          //save shape
        height = h;
        color = Color.black; //default color
    }
    //-----
    public void move(int x, int y) {
        xpos = x;          //save location
        ypos = y;
    }
    //-----
    public void setColor(Color c) {
        color = c;
    }
    //-----
    public void draw(Graphics g) {
        g.setColor(color); //set to curent color
        g.drawRect(xpos,ypos, width, height);
    }
}
```

Note that in the constructor, we set the default color of the rectangle to blue:

```
color = Color.black; //default color
```

by saving this color in the private **color** variable. Now, all rectangles will be black by default, although we could call the new **setColor** method to draw them in other colors.

Having added this method to the base **Rectangl** class, we now want to make a **Square** class that will always draw in red and with lines that are twice as thick. In order to do this, we must somewhere always make sure that the **setColor** method is called before drawing the square. The simplest way is simply to call this method in the square's constructor:

```
public Square2(int side) {
    super(side, side); //sides are the same size
    setColor(Color.red); //all squares are red
}
```

But how do we go about making thicker lines? It turns out that in version 1.1 of the Java window manager class, there is no line thickness method. So, the simplest way to draw a thicker square is to draw it, transpose it one pixel in both directions and draw it again. The only way we could manage to do this is to override the rectangle's **move** method to keep a copy of the x and y-coordinates and then to override the rectangle's **draw** method to do this double drawing. Our complete **Square** class is show below.

```
public class Square2 extends Rectangl {
    private int xpos, ypos; //saved copies

//This square class is derived from Rectangl
    public Square2(int side) {
        super(side, side); //sides are the same size
        setColor(Color.red); //all squares are red
    }
//-----
    public void move(int x, int y) {
//overrides base move method to
//save copies of coordinates
        xpos = x;
        ypos = y;
        super.move(x, y); //call parent procedure
    }
//-----
    public void draw(Graphics g) {
        super.draw(g); //draw in original
        move(xpos+1, ypos+1); //transpose one pixel
        super.draw(g); //draw it again
        move(xpos-1, ypos-1); //move it back
    }
}
```

This program is called `Sqr2.java` in `\chapter6` on the example disk. When you execute it, the rectangle and square display appears as shown in Figure 6-3.

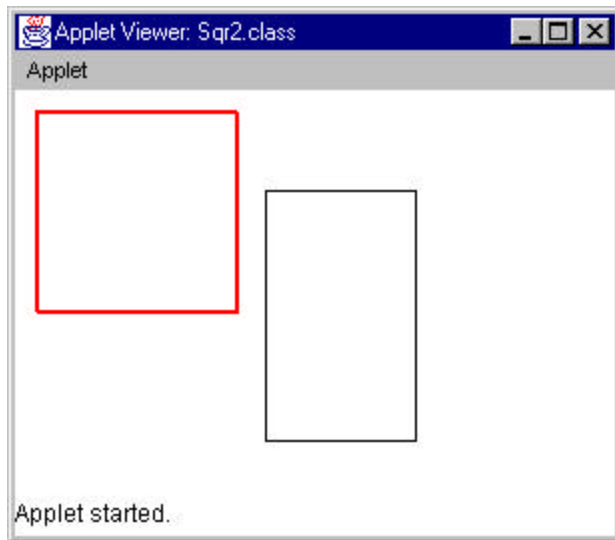


Figure 8-3 The Sqr2 program, showing a square drawn with double the line thickness of the rectangle.

Understanding public, private and protected

A **public** method or variable can be accessed by any code inside or outside the class. You can access a public method **foo()**, by calling **a.foo()** and a public variable **bar** by referring to **a.bar**.

A **private** method or variable can only be accessed within the class. You can call these methods directly within the class, but cannot call them or refer to private variables outside the class.

There are also three other visibility levels for variables: **private protected**, **protected** and *default* if no keyword is specified. A **private protected** method or variable can only be accessed within that class and classes derived from it. A **protected** method or variable is visible not only in derived classes but in all other classes in the same **package**. Packages are convenient ways of grouping related classes and allowing the reuse of class names. The **import** statements that start all of our programs are importing the contents of packages.

Methods and variables which are not marked as public, protected, or private are not nearly as well hidden as **private** methods. These methods are visible within the **package** but not within derived classes. Thus, other classes in the same package can refer to them almost as if they were public. Thus, it is always advisable to mark your methods and variables **private** unless you intend them to be public.

Inheriting “Event Methods”

In our simple example above, we used the **paint** method. In fact, we overrode the default paint method with one in our own program class to carry out some specific drawing. The paint method is one of the methods which you never call directly, but which is called by the window manager whenever the window needs to be repainted. These methods are essentially the wrapping of events in the windowing system. If you include these events in your program, it is important that they have the exact signature (calling arguments) shown in the reference guide, since this is the only way the Java runtime system can know to call them over the equivalent system events.

The paint method, for example, always has the form

```
public void paint(Graphics g)
```

where **g** represents the current Graphics object in use by the windowing system.

Other common methods include the **mouseDown** and **mouseUp** methods, which again, you never call, but are called by the system in response to user actions. These methods are really the encapsulation of mouse events and each has a default action. If you choose, you can derive a new method within your class for any of these event methods and they will be called first.

Common Event Methods
gotFocus
keyDown
keyUp
mouseDown
mouseDrag

mouseEnter
mouseLeave
mouseUp
paint

We'll see examples of overriding many of these event methods after we have formally taken up the Abstract Windows Toolkit (awt).

Abstract Classes

As you begin to design larger projects, you may find that you'd like to define the behavior of a class without writing the code for a specific method. For example, all shapes have an area, but in the basic Shape class, it is pointless to define an **area()** method, since each kind of shape will require a different sort of calculation.

Instead, you might choose to define an *abstract* shape class in order to define the methods you expect all shapes to be able to carry out.

```
abstract class Shape {
    public double area();
    public double circumference();
}
```

Then we can create a Rectangl class which inherits from this basic Shape class:

```
public class Rectangl extends Shape {
    //etc.
}
```

It is important to note, however, that if you say that your class is derived from an abstract class, you *must* provide methods for every method defined in the abstract class. If you don't, your new class is also treated as abstract and you won't be able to create instances of that class.

In this example, the Rectangl class we create must have methods for computing the area and circumference:

```
public class Rectangl extends Shape {
```

```

public double area()    {
    return width * height;
}
public double circumference() {
    return 2 * width + 2 * height;
}
}

```

Interfaces

Interfaces are another special kind of class definition: a class without any code associated with it. Since Java does not allow multiple inheritance, in which objects could inherit from two sets of parents, interfaces provide a way to create a set of classes which have rather different purposes but a few similar methods. If you say that a particular class *implements* an interface it is a promise that you have included methods in your class corresponding to each of the methods in the definition of the interface. We'll see specific examples of interfaces when we discuss layout managers in Chapter 8 and filename filters in Chapter 12.

For now, let's assume that we wanted to have a class Squasher that contained a method **squash** that we wanted to apply to our square class, as well as to other shape classes we might develop, like oval or circle. We don't even have to specify what this method does, only that it exists:

```

public interface Squasher {
    public void squash(float percent);
}

```

Note that this looks just like a class, except that the **squash** method contains no code, and the keyword **interface** replaces the keyword **class**.

Now, let's suppose we want to redefine our square class to use this method. We could declare it as

```

public class Sqr extends Rectangl implements Squasher

```

Now we are saying that we promise that we will include a method **squash** in this class and it will do whatever that method is supposed to do.

Interfaces Used in the Java Classes

Interfaces are used all throughout Java as a way of providing the same functionality to a wide variety of classes. You will find that all of the `EventListener` classes are interfaces to be implemented, but the `Iterator` and `Enumeration` interfaces occur in many of the non-visual classes as well.

The Enumeration Interface

For example, the `Enumeration` interface is just

```
interface Enumeration {
    public boolean hasMoreElements();
    public Object nextElement();
}
```

Any class that contains methods of these names is said to *implement* the `Enumeration` interface. We can quickly discover that the `StringTokenizer` class implements this interface.

The `Vector`, `Hashtable` and `HashMap` classes all have an `elements` method which returns an `Enumeration` object.

```
Vector v = new Vector();
//... some code adding items to the Vector
Enumeration enum = v.elements();
while enum.hasMoreElements() {
    String s = (String)enum.nextElement();
    System.out.println(s);
}
```

The Iterator Interface

The `Iterator` interface was added in Java 2 and has the same effect and style as the `Enumeration` interface, but has simpler method names and adds the `remove` method.

```
public boolean hasNext();
public Object next();
public void remove();
```

The `Vector`, `Hashtable` and `HashMap` classes all have an `iterator` method which returns an iterator of their contents:

```
Vector v = new Vector();
//... some code adding items to the Vector
```



```
Iterator itr = v.iterator();
while itr.hasMore() {
    String s = (String)itr.next();
    System.out.println(s);
}
```

Summary

In this chapter, we finally covered the last major piece of object-oriented programming: the use of inheritance to create new classes derived from existing classes. Once we derive a new class from an existing one we can add more function and override specific methods to give the original object more features.

In fact, you should think for a moment about how we derived and built the **Square2** class. Was editing the **Rectangl** class the best way to do this or should we have derived a **Rectangl2** class which had the color features we wanted?

We also looked briefly at the concepts of abstract classes and interfaces and how we can use them to give more common functionality to a group of unrelated classes.

In the next chapter we'll look in detail at the visual controls available in the awt windowing toolkit.

9. Java Visual Controls

Most Java programs are visual programs. Your Java programs are frequently visual interfaces to file and network processes as well as ways of entering data into interactive web pages. While Java really grew out of the Unix world, it has become extremely popular for Windows as well as on most other common operating system platforms, including Solaris, Macintosh systems, AIX and Linux. The visual controls are primarily those common to all of these platforms, although it is not difficult to write additional controls directly in Java.

The fundamental visual controls are

- TextField - a single line text entry field
- TextArea - a multiple line text entry field
- Checkbox - a combination of checkbox and Radio (Option) buttons
- List - a list box
- Button - a simple push button
- Choice - a dropdown list control
- Menu - a drop down menu from the window's toolbar
- Scrollbar - horizontal and vertical scrollbars.
- Panel - an area where you can group controls or paint images or graphics
- Canvas - a base class for creating your own controls.

All of these controls are part of the **java.awt** class, where "awt" stands for one or more of the following:

- a window toolkit
- advanced window toolkit

- another window toolkit
- abstract window toolkit, and by critics it is called the
- awful window toolkit

As we noted, the toolkit is an intersection of the window functions found on all of the common windowing systems. Since the controls on each platform are actually implemented using the native platform functions, they look somewhat different on each platform, but have the same logical properties on each.

By contrast, the Java Foundation (or Swing) classes are implemented entirely in Java. They are better looking and more sophisticated, but are not supported in all browsers and are somewhat harder to program with. We'll take them up in a following chapter.

The Visual Class Hierarchy

All visual controls are children of the **Component** class which is in turn derived from the **Object** class. All objects in Java are derived directly or indirectly from this class. Therefore all of the methods of the **Object** and **Component** classes are automatically available to any of the visual control classes. This is illustrated in Figure 7-1.

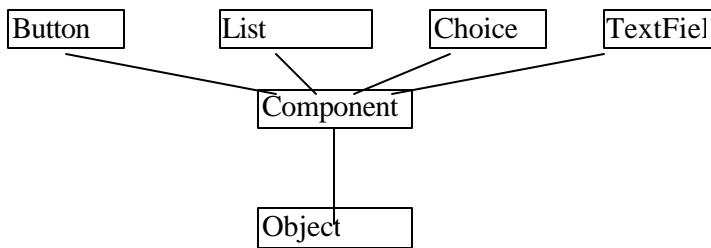


Figure 7-1: Some of the visual classes in the object inheritance tree.

Some of the common methods in these base classes include

paint	called to redraw control
setBounds(x, y, w, h)	change control shape and location
setSize(w, h)	change control size
setEnabled(boolean)	set control to active or inactive
getBackground	get background color
setBackground(Color)	set background color
setVisible(boolean)	set control visibility
gotFocus	called if control receives focus
isEnabled	returns boolean whether currently enabled
isVisible	returns boolean whether control is visible
keyDown	called if character key pressed
keyUp	called if character key released
getLocation	returns x and y location in Point object
setLocation(x, y)	moves the control to new location
repaint	causes control to be repainted
requestFocus	asks system to give focus to control
setSize(w, h)	changes size of control
setBackground(Color)	sets background color
setForeground(Color)	sets foreground color
setFont(String, int, int)	sets font to named font, style and size
getSize	returns current size of control

In every case, these methods are ones which have default results built into the Java system. However, if you write your own version of these methods, having the exact same calling sequence as that of the standard methods, you are *overriding* or *subclassing* these methods and replacing them with your own. Often the action you want to take is to do some small thing, and then have the system defined default action take place as well. You do this by calling the same method in the parent class using **super** to represent that class:

```
//derived resize method
//saves a local copy of the width and height
int w, h;
```

```

public void setSize(int width, int height) {
    w = width;                //Save a local copy
    h = height;
    super.setSize(width, height);    //call parent
}

```

Event Driven Programming

All visual controls are children of the *Component* class, which is in turn derived from the *Object* class. All objects in Java are derived directly or indirectly from this class. Therefore, all of the methods of the *Object* and *Component* classes are automatically available to any of the visual control classes. This is illustrated in Figure 9-1. Table9-1 includes some of the common methods in the *Object* and *Component* classes.

Figure 9-1: Some of the visual classes in the object inheritance tree.

Method	Description
paint	Called to redraw control.
setBounds(x, y, w, h)	Changes control shape and location.
setSize(w, h)	Changes control size.
setEnabled(boolean)	Sets control to active or inactive
getBackground	Gets background color.
setBackground(Color)	Sets background color.
setVisible(boolean)	shows or hides control.
isEnabled	Returns boolean whether currently enabled.
isVisible	Returns boolean whether control is visible.
getLocation	Returns x and y location.
setLocation(x, y)	Moves the control to new location.
repaint	Causes control to be repainted.
requestFocus	Asks system to give focus to control.
setSize(w, h)	Changes size of control.
setBackground(Color)	Sets background color.
setForeground(Color)	Sets foreground color.

`setFont(String, int, int)` Sets font to named font, style, and size.

`getSize` Returns current size of control.

Table 9-1: Common methods in the Component class

Event-Driven Programming

Java is an *event-driven* programming language. Since the programs exist in a windowing environment, the display is affected by events such as mouse movements and clicks, key presses, window resizing, and window rearrangement. For a program to behave as expected in such an environment, it must be receptive to all of these events and take appropriate action.

Java defines a package called `java.awt.event` which contains a set of interfaces and methods for dealing with events. These event classes are shown in Table 9-2:

```

ActionEvent
AdjustmentEvent
ComponentAdapter
ComponentEvent
ContainerAdapter
ContainerEvent
FocusAdapter
FocusEvent
InputEvent
ItemEvent
KeyAdapter
KeyEvent
MouseAdapter
MouseEvent
MouseMotionAdapter
PaintEvent
TextEvent
WindowAdapter
WindowEvent

```

Table 9-2: The Event interface classes provided in Java 1.1 and later.

Controls can register an interest in such events using the methods

```

obj.addActionListener(Listener);
obj.addItemListener(Listener);

```

```
obj.addMouseListener(Listener);
obj.addMouseMotionListener(Listener);
obj.addKeyListener(Listener);
obj.addFocusListener(Listener);
obj.addComponentListener(Listener);
```

where the *Listener* class is frequently the current class,

```
obj.addActionListener(this);
```

but can be an instance of any other class as well:

```
obj.addActionListener(catchClass);
```

The class that catches these events need not poll for all possible events but only for those in which it has declared an interest. Further, each of these event types calls a specific routine. For example, the *Button* class can add an action listener:

```
Button bt = new Button("OK");
bt.addActionListener(this);      //this class looks for click
// . . .
public void actionPerformed(ActionEvent evt)
//This method receives the click for the button
```

This is summarized in Table 9-3.

Control	Registers Interest	Receives Event
Button List MenuItem TextField	addActionListener	actionPerformed
Checkbox Choice List Checkbox- MenuItem	addItemListener	itemStateChanged
Dialog Frame	addWindowListener	windowClosing windowOpened windowIconified windowDeiconified windowClosed windowActivated windowDeactivated
Dialog Frame	addComponentListener	componentMoved componentHidden componentResized

		componentShown
Scrollbar	addAdjustmentListener	adjustmentValueChanged
Checkbox Checkbox- MenuItem Choice List	addItemListener	itemStateChanged
Canvas Dialog Frame Panel Window	addMouseListener	mousePressed mouseReleased mouseEntered mouseExited mouseClicked
Canvas Dialog Frame Panel Window	addMouseMotionListener	mouseDragged mouseMoved
Component	addKeyListener	keyPressed keyReleased keyTyped
Component	addFocusListener	focusGained focusLost
TextComponent	addTextListener	textValueChanged

Table 9-3: Java components, listener classes and listener methods.

Action Events

Action events can occur when a button, list box, or text box is clicked on, if your control registers an interest in those events:

```
Button bt = new Button("OK");
bt.addActionListener(this);
```

At the same time, it is indicating that the `ActionListener` class that is to receive these events is the current class. Now where is this class? Well it turns out that `ActionListener` is an *interface* rather than an actual class. So we merely need to tell the class containing our button that it *implements* the `ActionListener` interface.

```
public class myButtons extends Frame
implements ActionListener {
}
```


Then, somewhere in that class we must include all of the methods of the `ActionListener` interface: in this case one method:

```
public void actionPerformed(ActionEvent aEvt)
```

If there is only one control on which we have called the method `addActionListener` we don't even have to check to see which control called the action method, we just carry out the button's purpose. If there is more than one such control, we need to check to see which one called this method:

```
public void actionPerformed(ActionEvent aEvt)
{
    Object source = aEvt.getSource();
    if (source == bt)
        System.out.println("button pressed");
    else
        System.out.println("other control did it");
}
```

Unlike the Java event handling in version 1.0, this method is *only* called by controls if you have specifically called their `addActionListener` method. Thus, in many cases you may not need to test for other controls as we did above.

The Controls Demonstration Program

The `ControlDemo` demonstration program illustrates all of the controls we discuss in this chapter. The source is provided as `ControlDemo.java` in the `\chapter7` directory on the Companion CD-ROM. The list box shows a record of all of the events on the other controls, and it can be cleared using the Clear button. The TextField edit box allows you to type in characters if the L(ocked) check box is not checked. If the P(assword) check box is checked, the characters are echoed as asterisks. The label above the edit field changes depending on the state of these two check boxes. It also changes when you click on an entry in the list box.

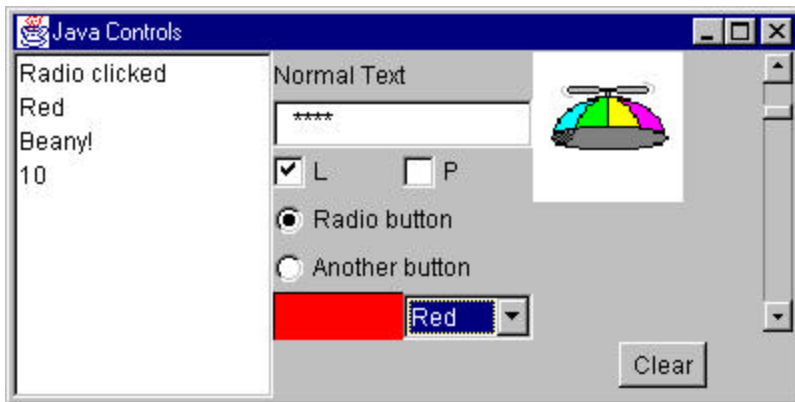


Figure 9-2: The Controls demonstration program.

You can click on either of the two radio buttons and have its label appear in the list box. There are two panels below the radio buttons, one flashing between blue and green and one displaying a beanie. The dropdown list or Choice box, a drop-down list box, allows you to switch between having blue-green and red-green flashing. If you click on the beanie, its name appears in the list box. The program window is shown in Figure 9-2.

Rather than going over the Controls demonstration program in this chapter, we will introduce the controls in this chapter and show their use in the chapters that follow. However, you can examine the source code of this program on the companion CD-ROM.

The Button Control

The button control can have a label and can be clicked on with the mouse. It also can be enabled, disabled, shown, hidden, and have its caption changed. This basic button cannot have its background color changed (in Windows 95) or display an image, but we will see that these features have been added in the Swing controls. The methods for setting and reading the button labels are described in Table 9-4, but you usually set the button's text as part of the constructor:

```
Clearit = new Button("Clear");
```

Method	Description
setLabel(String)	Sets the button's label.
getLabel	Retrieves the button's label.

Table 9-4: Methods for the Button class

The Label

The label is simply a place to display static text. It has only two constructors:

```
Label lbl = new Label();           //create an empty label
Label lbl = new Label(String, align); //create label with text
//align can be LEFT, CENTER or
RIGHT
```

Method	Description
setAlignment(int)	Sets label alignment.
setText(String)	Sets label text.
getText	Returns label text.

Table 9-5: Methods for the Label control

The label methods are equally simple (they are shown in Table 9-5). Note that you can set the font and color of a label at any time in your program.

TextFields & TextAreas

A TextField is a single line where you can type in text, and a TextArea is a multiline entry field. The constructors are:

```
//create text field with string displayed
TextField tf = new TextField(String);
//create empty text field n characters wide
TextField tf = new TextField(n);

//create text area of spec'd # of rows and columns
TextArea ta = new TextArea(rows, cols);
```

The most significant methods for these controls are shown in Table 7-6.

Method	Description
---------------	--------------------

TextArea

<code>getText</code>	Returns current text in box.
<code>setText(String)</code>	Sets text field to that string.
<code>setEditable(boolean)</code>	Sets whether text can be edited.
<code>select(start, end)</code>	Selects the text characters specified.
<code>selectAll</code>	select all the text

TextField

<code>setEchoChar(char)</code>	Set character to be echoed to allow password entry. To undo this, set the echo char to <code>'\0'</code> .
--------------------------------	--

Table 9-6: TextArea and TextField methods.

Both `TextArea` and `TextField` are classes derived from `TextComponent`, and as you can see from Table 9-3, they can receive `textValueChanged` events as well as the `keyPress` and `focus` events.

The List box

A List box is a vertical list of single lines of text. You can add to it, select or change items, and delete items. If you add more items than can be displayed, a scroll bar appears on the right side.



There are two constructors for the list control:

```
//create new list with no visible rows
List list1 = new List();
```

```
//create new list with n visible rows
//and whether to allow multiple selections
List list1 = new List(n, boolean);
```

The important methods are described in Table 9-7.

Method	Description
<code>addItem(String)</code>	Adds an item to the end of the list.
<code>addItem(String, n)</code>	Adds an item at position <i>n</i> in the list.
<code>removeAll()</code>	Clears the list (but for Win95, see below).
<code>remove(int n)</code>	Removes item <i>n</i> from the list
<code>remove(String s)</code>	Removes first item matching String <i>s</i>
<code>getItemCount</code>	Returns number in list.
<code>deselect(n)</code>	Deselects item <i>n</i> .
<code>getSelectedIndex</code>	Returns index of selected item.
<code>getSelectedItem</code>	Returns text of selected item.
<code>getItem(n)</code>	Returns text of item <i>n</i> .
<code>isSelected(n)</code>	Returns true if item is selected.
<code>replaceItem(n,String)</code>	Replaces item <i>n</i> with next text.
<code>select(n)</code>	Selects item <i>n</i> .
<code>setMultipleSelections (boolean)</code>	Sets list to allow or not allow

multiple selections.

Table 9-7: Important listbox methods.

In addition to adding items to list boxes and seeing what line or lines are selected, you might want to change some program display element when the user selects a line in a list box. If you call the `addItemListener` method, it causes the `itemStateChanged` method to be called whenever you click on a line in a list box. Calling the `addActionListener` method causes the `actionPerformed` method to be called whenever you *double* click on a list box element.

The Choice Box

The Choice box is a single-line window with a drop-down arrow revealing a drop-down list box.



It has the same constructors and methods as the List box. The events for a Choice box are slightly different: selecting an item from a Choice box generates an `actionPerformed` event, while clicking on an item in a list box generates an `itemStateChanged` event.

The Choice box is not a Windows-style combo box where you can type in or select from the top line, but you could easily construct such a combo box from a text field, a button, and a hidden list box.

The Scroll Bar

While list boxes contain their own scroll bars, it is sometimes useful to have scroll bars for selecting other kinds of variable input. The Scrollbar control can be constructed as a horizontal or vertical scroll:

```
scroller = new Scrollbar(orient); //HORIZONTAL or VERTICAL
```

```
scroller = new Scrollbar(orient, value, visible, min, max);
```

You register an interest in Scrollbar events by calling `addAdjustmentListener` and receive events in the `adjustmentValueChanged` method. There are several possible events that results in this method being called

<code>UNIT_INCREMENT</code>	the scroll bar has been clicked at the top
<code>UNIT_DECREMENT</code>	or bottom to move one unit up or down
<code>BLOCK_INCREMENT</code>	the scroll bar has been click just above
<code>BLOCK_DECREMENT</code>	or below the elevator causing a “page up” or “page down” event
<code>TRACK</code>	the elevator has been dragged to a new position

You can determine which from within this method as follows:

```
public void adjustmentValueChanged(AdjustmentEvent aEvt)
{
    switch(aEvt.getAdjustmentType())
    {
        case AdjustmentEvent.UNIT_DECREMENT:    //etc..
        case AdjustmentEvent.TRACK:             //etc..
    }
}
```

The Scrollbar class methods you can use are explained in Table 9-8.

Method	Description
getValue()	Returns position of scroll relative to min and max.
setValue(int)	Sets position of slider.
setValues(val, posn, min, max)	Sets parameters for scroll bar.
setUnitIncrement(int)	Sets how much to move on one click.
setBlockIncrement(int)	Sets how much to move on page up/down increment.

Table 9-8: Scroll bar methods.

Checkboxes

A check box is a square box that you can click on or off. However, unlike some other Windows check boxes, it does not have a third grayed-out state. Check boxes operate independently from each other: you can check or uncheck as many as you like. When a check box is checked, it returns as state of true; if unchecked it returns false. The important methods are described in Table 9-9. The constructors are:

```
//create checkbox with label
Checkbox cb = new Checkbox(String);
```

Method	Description
getState	Returns state of checkbox.
setState(boolean)	Sets state of checkbox.
getLabel	Gets label of checkbox.

`setLabel(String)` Sets label for checkbox.

Table 9-9: Important Checkbox methods.

If a user clicks on a checkbox, and you have registered the `addItemListener` method, then you can receive notification in the `itemStateChanged` event of both checking and unchecking of the boxes. If you know which box caused the event notification, then you can simply ask whether the checkbox is currently checked or not to decide what action to take. You can also execute the `getStateChange` method on the `itemEvent`, which will return `DESELECTED` or `SELECTED`.

```
public void itemStateChanged(ItemEvent iEvt)
{
    int evType = iEvt.getStateChange();
    if (evType == ItemEvent.SELECTED)
        //blah
    else
        //blah blah
}
```

Radio Buttons

Radio buttons are in fact a special case of check boxes in Java. Radio buttons have all the same methods as check boxes, but appear as circles where only one of a group can be selected if the check box items are made members of a `CheckboxGroup`.

Java allows you to have several groups of radio buttons on a page, as long as you make each set members of a different `CheckboxGroup`.

The following Java statements create a pair of radio buttons belonging to a single check box group:

```
//First create a new Check box group
CheckboxGroup cbg = new CheckboxGroup();

//then create check boxes as part of that group
Checkbox Female = new Checkbox("Female", cbg, true);
Checkbox Male = new Checkbox("Male", cbg, false);
```


Note that it is the use of this kind of `Checkbox` constructor that causes the boxes to be displayed as rounded radio buttons. Like ordinary checkboxes, you can register an interest with `addItemListener`.

Colors

Colors in Java can be specified as RGB hexadecimal numbers or as named constants. Each of the red, green, and blue color components can vary between 0 and 255. You usually use them in `setBackground` and `setForeground` statements, and often using the predeclared constants in Table 7-10.

black
blue
cyan
darkGray
grey
green
lightGray
magenta
orange
pink
red
yellow
white

Table 7-10: Predeclared color constants.

Since these constants are all part of the `Color` class, you refer to them using the class name:

```
setBackground(Color.blue);
```

You can also create specific colors using a single integer to represent the RGB values or by using a group of three byte values:

```
setForeground(new Color(0xff00dd));  
setBackground(new Color(255, 0, 0xdd));
```

Fonts

There are five basic fonts that Java supports on all platforms:

- TimesRoman
- Helvetica
- Courier
- Dialog
- DialogInput

For any given platform, you may also use any font that you know is available.

The default font for Java applets and applications is Roman. Since sans serif fonts are more readable for window labels and text, you should always switch the font of any class containing visual controls to Helvetica using the `setFont` method:

```
setFont(new Font("Helvetica", Font.PLAIN, 12));
```

where the styles may be:

```
Font.PLAIN  
Font.BOLD  
Font.ITALIC, or  
Font.ITALIC + Font.BOLD.
```

FontMetrics

The `FontMetrics` class contains methods to find out the size of the current font or the width of a string drawn in that font. It is not related to or contained in the `Font` class itself, but is instead dependent on where you are drawing the font. The `Graphics` class contains the call `getFontMetrics()`, which returns an instance of the class for that graphics object. For this reason, you can only determine the actual size of a font with a `paint` or related method, where the current graphics context is available.

Summary

In this chapter, we've looked at the set of visual controls that the Java abstract window toolkit (awt) uses. We've listed their events and their methods and talked a little about how event-driven programming works.

You can't really appreciate how to use these controls to build real programs unless we take you through some examples. We'll do so in the next couple of chapters.

