

# JAVA

---

**“Aim at the stars, even if you fall, you will fall on the moon.....”**

**-By Mr. Bhavin Rawal**



**JAVA**

<b>Introduction</b> .....	<b>7</b>
<b>History</b> .....	<b>8</b>
<b>Characteristics</b> .....	<b>10</b>
<b>OOP</b> .....	<b>10</b>
<b>Simple</b> .....	<b>10</b>
<b>Portable</b> .....	<b>11</b>
<b>Robust</b> .....	<b>11</b>
<b>Multithreaded</b> .....	<b>12</b>
<b>Architecture-Neutral</b> .....	<b>12</b>
<b>Compiled – Interpreted</b> .....	<b>13</b>
<b>High Performance</b> .....	<b>13</b>
<b>Distributed</b> .....	<b>13</b>
<b>Dynamic</b> .....	<b>13</b>
<b>Java Virtual Machine</b> .....	<b>14</b>
<b>2.Language Fundamental</b> .....	<b>14</b>
<b>First Program</b> .....	<b>14</b>
<b>Compiling a program</b> .....	<b>16</b>
<b>Compile the Source File</b> .....	<b>16</b>
<b>Running a program</b> .....	<b>18</b>
<b>In the same directory, enter at the prompt:</b> .....	<b>18</b>
<b>Now you should see:</b> .....	<b>18</b>
<b>Command line argument</b> .....	<b>18</b>
<b>Accepting user inputs</b> .....	<b>19</b>
<b>Datatypes</b> .....	<b>19</b>
<b>Integer Types</b> .....	<b>19</b>
<b>Floating-Point Types</b> .....	<b>22</b>
<b>Characters</b> .....	<b>23</b>
<b>Boolean</b> .....	<b>24</b>
<b>Literals</b> .....	<b>24</b>
<b>Variables</b> .....	<b>25</b>
<b>Declaration</b> .....	<b>25</b>
<b>Dynamic Initialization</b> .....	<b>26</b>
<b>Scope</b> .....	<b>28</b>
<b>Automatic Type Promotion in Expression</b> .....	<b>30</b>
<b>Operators</b> .....	<b>31</b>
<b>Arithmetic</b> .....	<b>31</b>
<b>Relational and Logical</b> .....	<b>33</b>
<b>Assignment</b> .....	<b>33</b>
<b>Conditional / Ternary</b> .....	<b>35</b>
<b>Increment – Decrement</b> .....	<b>36</b>
<b>Bitwise Shift</b> .....	<b>37</b>
<b>Operator Precedence</b> .....	<b>38</b>

<b>Control Statements .....</b>	<b>40</b>
<b>If ... else .....</b>	<b>40</b>
<b>Switch ... case.....</b>	<b>41</b>
<b>Loops .....</b>	<b>42</b>
<b>Nested Loops and Labeled Loops.....</b>	<b>46</b>
<b>Jump Statements.....</b>	<b>48</b>
<b>Arrays.....</b>	<b>51</b>
<b>3. Implementing Classes .....</b>	<b>55</b>
<b>Basics.....</b>	<b>55</b>
<b>General Form of a Class.....</b>	<b>56</b>
<b>Creating Classes &amp; their Objects .....</b>	<b>57</b>
<b>Object.....</b>	<b>59</b>
<b>Assigning Object Reference Variable .....</b>	<b>60</b>
<b>Methods.....</b>	<b>61</b>
<b>Constructors .....</b>	<b>62</b>
<b>The <i>this</i> keyword .....</b>	<b>64</b>
<b>Garbage Collection .....</b>	<b>65</b>
<b>Overloading .....</b>	<b>65</b>
<b>Understanding final &amp; static.....</b>	<b>66</b>
<b>Nested / Inner Classes.....</b>	<b>67</b>
<b>4. Inheritance.....</b>	<b>69</b>
<b>Basics .....</b>	<b>69</b>
<b>extends keyword declares that your class is a subclass of another .....</b>	<b>70</b>
<b>Modifiers.....</b>	<b>73</b>
<b>The static Modifier.....</b>	<b>74</b>
<b>The final Modifier .....</b>	<b>75</b>
<b>The synchronized Modifier .....</b>	<b>75</b>
<b>The native Modifier .....</b>	<b>76</b>
<b>The <i>super</i> keyword.....</b>	<b>76</b>
<b>Constructor's hierarchy .....</b>	<b>77</b>
<b>Overriding .....</b>	<b>79</b>
<b>Methods.....</b>	<b>79</b>
<b>Calling the Overridden Method .....</b>	<b>80</b>
<b>Variables .....</b>	<b>80</b>
<b>Abstract Classes .....</b>	<b>81</b>
<b>Abstract Method .....</b>	<b>82</b>
<b>Using <i>final and Static</i> .....</b>	<b>83</b>
<b>5.Packages &amp; Interfaces .....</b>	<b>84</b>
<b>Packages.....</b>	<b>84</b>
<b>Defining a package.....</b>	<b>84</b>
<b>Interfaces .....</b>	<b>86</b>
<b>Basics.....</b>	<b>86</b>
<b>Interface References .....</b>	<b>86</b>
<b>Applying Interfaces.....</b>	<b>87</b>

Interface variables .....	90
Interface Inheritance .....	90
<b>Exception Handling.....</b>	<b>91</b>
Basics .....	91
Exception & Error Classes.....	92
Class Throwable.....	92
Errors.....	92
Exceptions.....	93
Checked Exceptions .....	93
Using try ... catch .....	94
Multiple Catch Statements.....	97
Throw, Throws & Finally Statements .....	97
The finally Block.....	99
Java's Built in Exception.....	100
<b>Java.lang.* .....</b>	<b>101</b>
Interfaces .....	101
Cloneable .....	101
Comparable .....	101
Runnable.....	102
Classes .....	103
Object.....	103
Number .....	104
Wrapper Classes .....	104
Class .....	106
Math .....	106
String and StringBuffer.....	108
String Access Methods.....	109
String Access Methods.....	109
String Conversion and Generation.....	110
System .....	113
Thread.....	115
ThreadGroup.....	116
Throwable.....	116
The Error Class.....	116
The Exception Class.....	117
<b>Java.util.* .....</b>	<b>117</b>
Basics .....	117
Interfaces .....	118
Collection .....	118
List.....	119
Set .....	119
SortedSet.....	119
Map.....	119
SortedMap .....	120
Enumeration and Iterator .....	120

<b>Classes .....</b>	<b>120</b>
All Implementing Classes of Collection & Map .....	120
Properties.....	130
StringTokenizer.....	133
Date.....	135
Calendar.....	137
Random .....	139
<b>Multithreading .....</b>	<b>140</b>
Basics.....	140
<b>Customizing a Thread's run Method.....</b>	<b>141</b>
Subclassing Thread and Overriding run .....	141
Implementing the Runnable Interface.....	144
Deciding to Use the Runnable Interface.....	145
<b>The Life Cycle of a Thread .....</b>	<b>146</b>
Creating a Thread.....	147
Starting a Thread.....	148
Making a Thread Not Runnable.....	148
Stopping a Thread.....	149
The isAlive Method.....	151
<b>Understanding Thread Priority.....</b>	<b>151</b>
Time-Slicing.....	152
Summary.....	155
<b>Synchronizing Threads.....</b>	<b>155</b>
<b>The Producer/Consumer Example.....</b>	<b>156</b>
The Main Program .....	160
The Output .....	161
<b>Locking an Object.....</b>	<b>161</b>
<b>Requiring a Lock .....</b>	<b>163</b>
<b>Using the notifyAll and wait Methods .....</b>	<b>164</b>
<b>Avoiding Starvation and Deadlock .....</b>	<b>167</b>
<b>Grouping Threads.....</b>	<b>168</b>
The Default Thread Group .....	169
Creating a Thread Explicitly in a Group.....	169
Getting a Thread's Group.....	170
The ThreadGroup Class.....	170
Collection Management Methods.....	170
Methods that Operate on the Group.....	171
Methods that Operate on All Threads within a Group.....	174
Access Restriction Methods .....	174
<b>Summary.....</b>	<b>176</b>
<b>7. IO .....</b>	<b>177</b>
<b>Basics.....</b>	<b>177</b>
Character Streams .....	178
Byte Streams.....	179
<b>Understanding the I/O Superclasses .....</b>	<b>180</b>

<b>Using the Streams .....</b>	<b>181</b>
<b>Understanding the Implementation of various IO classes .....</b>	<b>184</b>
FileInputStream, FileOutputStream, FileReader, FileWriter.....	184
ByteArrayInputStream, ByteArrayOutputStream .....	187
InputStreamReader, OutputStreamWriter .....	188
BufferedInputStream, BufferedOutputStream, BufferedReader, BufferedWriter ...	189
DataInputStream, DataOutputStream .....	191
SequenceInputStream.....	194
CharArrayReader, CharArrayWriter.....	195
File, RandomAccessFile.....	197
PrintWriter .....	198
PushbackInputStream, PushbackReader.....	199
ObjectInputStream, ObjectOutputStream.....	200
<b>AWT Fundamentals.....</b>	<b>202</b>
Graphical User Interfaces .....	202
AWT Basics .....	202
Applications versus Applets.....	202
Basic GUI Logic .....	203
A Simple Example.....	203
AWT Components .....	205
Buttons .....	205
Canvas.....	206
Checkbox .....	207
CheckboxGroup .....	208
Choice.....	209
Label.....	209
List.....	210
Scrollbar.....	211
TextField .....	211
TextArea .....	212
Common Component Methods.....	214
<b>Containers.....</b>	<b>216</b>
Common Container Methods.....	216
ScrollPane .....	217
<b>Event Handling .....</b>	<b>218</b>
Events.....	218
AWTEvent.....	220
Event Sources .....	221
Event Listeners.....	222
Summary of Listener interfaces and their methods .....	223
Event Adapters.....	224
Adapters Example.....	226
<b>Applications and Menus.....</b>	<b>227</b>
GUI-based Applications .....	227
Applications: Dialog Boxes.....	230

Applications: Menus .....	233
Menu Shortcuts .....	237
Pop-up Menus.....	238
<b>APPLET .....</b>	<b>240</b>
<b>Overview of Applets.....</b>	<b>240</b>
A Simple Applet .....	241
Loading the Applet .....	243
Leaving and Returning to the Applet's Page.....	243
Reloading the Applet .....	243
Quitting the Browser .....	244
Summary.....	244
<b>Methods for Milestones .....</b>	<b>244</b>
<b>Methods for Drawing and Event Handling .....</b>	<b>246</b>
<b>Methods for Adding UI Components.....</b>	<b>247</b>
Pre-Made UI Components .....	248
Methods for Using UI Components in Applets .....	249
Adding a Non-Editable Text Field to the Simple Applet .....	249
<b>Using the APPLETTAG .....</b>	<b>252</b>
Specifying Parameters .....	252
Specifying Alternate HTML Code and Text .....	254
Specifying the Applet Directory .....	256
<APPLET> Tag Attributes .....	257
<b>What Applets Can and Can't Do.....</b>	<b>260</b>
Security Restriction .....	260
Applet Capabilities.....	261
<b>Network Programming.....</b>	<b>262</b>
<b>The Internet Protocol Suite.....</b>	<b>262</b>
The Internet.....	262
Connection-Oriented Versus Connectionless Communication .....	265
<b>Sockets and Client/Server Communication.....</b>	<b>266</b>
Overview of java.net.....	268
The InetAddress Class .....	268
The Socket Class.....	270
The ServerSocket Class.....	273
The DatagramSocket Class .....	277
The DatagramPacket Class .....	278
URL .....	282

## Introduction

## History

Few years ago, James Gosling was part of Green, an isolated research project at Sun that was studying how to put computers into everyday household items. The researchers wanted to make smart appliances such as thoughtful toasters and lucid lamps. The group also wanted these devices to communicate with each other. With this vision, the Greens built a device called Star7. This was a handheld remote control operated by touching animated objects on the screen. A Star7 user could navigate by fingertip.

The most remarkable ability of the Star7 device was how it communicated with other Star7 devices. An on-screen object could be passed from one device to another. The original plan was for the Star7 operating system to be developed in C++. But Gosling didn't find it feasible, so he holed up in his office and wrote a new language that was better for the purposes of the Green project than C++. He called the language Oak in honor of a tree that could be seen from his office window.

The Green project had an impressive demonstration device, operating system, and programming language. Sun's higher-ups gave the green signal and the project was incorporated as FirstPerson in November 1992. But unfortunately this project failed.

Marc Andreessen, an undergraduate student working at the National Center for Supercomputing Applications, developed the first visual World Wide Web browser, Mosaic 1.0. This event had sparked an international phenomenon, and the Web was rapidly becoming a mass medium.

In mid-1994, the folks who stuck with Oak found their reason for being: the World Wide Web. When Oak was created, the Web was a little-known service bouncing around the high-energy physics community. The Oak technology was well suited for this medium, especially because of its ability to run on multiple platforms. More importantly, it



introduced something that wasn't available anywhere else—programs that could be run on user's computers safely from a Web page.

Patrick Naughton and Jonathan Payne finished WebRunner, a Web browser that brought back the star of the Star7, Duke. Sun realized it had something promising on its hands, but soon found that Oak could not be trademarked because a product was already using the name.

After brainstorming sessions in January 1995 to supplant the Oak name, Java won for the language and HotJava replaced WebRunner as the browser's name. Java was the name chosen because it sounded the coolest. It won out over DNA, Silk, Ruby, and WRL (WebRunner Language).

The project now had a cool name, a cool new purpose, and a HotJava browser to show it off. On March 23, 1995, it attracted a cool new admirer: that Andreesen kid. In a front-page story, the San Jose Mercury News reported that Sun was working on a project to make Web pages "as lively as a CD-ROM." The story included the following quote from Andreesen, who had become a vice president at Netscape (and had also become a self-contained Bill Gates starter kit): "What these guys are doing is undeniably, absolutely new."

The phenomenon was on. Netscape licensed the Java language for use in its browser a few months after the article ran, putting the language in front of millions of Netscape users. The first beta release of Java was made available for download in November 1995. Sun made a developer's kit and the source code for its product freely available to anyone who wanted it—and by that time, thousands of people and companies did.

Thus, SUN gifted a new object-oriented, made-for-the-Internet programming language **JAVA**.

## **Characteristics**

### **OOP**

**A software design method that models the characteristics of abstract or real objects using classes and objects.**

Java deals with classes and objects, pure and simple. They aren't just more data structures that are available to the programmer—they are the basis for the entire programming language.

In Java, classes and objects are at the center of the language. Everything else revolves around them. You can't declare functions and procedures. They don't exist. You can't use structures or unions. Java provides all the luxuries of object-oriented programming: class hierarchy, inheritance, encapsulation, and polymorphism—in a context that is truly useful and efficient. Once you have begun developing software in Java, you have two choices:

1. Build on the classes you have developed, thereby reusing them.
2. Rewrite your software from scratch.

With Java, the temptation to start from scratch is no longer appealing. Java's object-oriented structure forces you to develop more useful, and much simpler software.

### **Simple**

Java was modeled after C and C++. The object-oriented approach, and most of Java's syntax, is adapted from C++. Programmers who are familiar with that language (or with C) will have a much easier time learning Java because of the common features.

### **Secure**

Memory management occurs automatically in Java—programmers do not have to write their own garbage-collection routines to free up memory.

Because Java does not use pointers to directly reference memory locations, as is prevalent in C and C++, Java has a great deal of control over the code that exists within the Java environment.

It was anticipated that Java applications would run on the Internet and that they could dynamically incorporate or execute code found at remote locations on the Internet. Because of this, the developers of Java feared that Java compiler might generate Java byte codes with the intent of bypassing Java's runtime security. This led to the concept of a byte-code verifier. The byte-code verifier examines all incoming code to ensure that the code plays by the rules and is safe to execute. In addition to other properties, the byte code verifier ensures the following:

No pointers are forged.

No illegal object casts are performed.

There will be no operand stack overflows or underflows.

All parameters passed to functions are of the proper types.

Rules regarding private, protected, and public class membership are followed.

## **Portable**

Java code is portable. It was an important design goal of Java that it be portable so that as new architectures (due to hardware, operating system, or both) are developed, the Java environment could be ported to them.

In Java, all primitive types (integers, longs, floats, doubles, and so on) are of defined sizes, regardless of the machine or operating system on which the program is run. This is in direct contrast to languages like C and C++ that leave the sizes of primitive types up to the compiler and developer.

Additionally, Java is portable because the compiler itself is written in Java.

## **Robust**

Java was created as a strongly typed language. Data type issues and problems are resolved at compile-time, and implicit casts of a variable from one type to another are not allowed.

Memory management has been simplified in Java in two ways. First, Java does not support direct pointer manipulation or arithmetic. This makes it impossible for a Java program to overwrite memory or corrupt data. Second, Java uses runtime garbage collection instead of explicit freeing of memory. In languages like C++, it is necessary to delete or free memory once the program has finished with it.

## **Multithreaded**

Writing a computer program that only does a single thing at a time is an artificial constraint that we've lived with in most programming languages. With Java, we no longer have to live with this limitation. Support for multiple, synchronized threads is built directly into the Java language and runtime environment.

Synchronized threads are extremely useful in creating distributed, network-aware applications. Such an application may be communicating with a remote server in one thread while interacting with a user in a different thread.

## **Architecture-Neutral**

It is not easy to write an application that can be used on Windows NT, UNIX, and a Macintosh. And it's getting more complicated with the move of Windows NT to non-Intel CPU architectures.

Java takes a different approach. Because the Java compiler creates byte code instructions that are subsequently interpreted by the Java

interpreter, architecture neutrality is achieved in the implementation of the Java interpreter for each new architecture.

## **Compiled – Interpreted**

The Java compiler (`javac`) is the component of the Java Developer's Kit used to transform Java source code files into bytecode executables that can be run in the Java runtime system . It is the job of the Java compiler to process Java source code files and create executable Java bytecode classes from them. Executable bytecode class files have the extension `.class`, and they represent a Java class in its useable form.

The Java interpreter used to execute Java applications. The interpreter translates byte codes directly into program actions.

## **High Performance**

For all but the simplest or most infrequently used applications, performance is always a consideration for most applications, including graphics-intensive ones such as are commonly found on the World Wide Web, the performance of Java is more than adequate.

## **Distributed**

Java facilitates the building of distributed applications by a collection of classes for use in networked applications. By using Java's URL (Uniform Resource Locator) class, an application can easily access a remote server. Classes also are provided for establishing socket-level connections.

## **Dynamic**

Because it is interpreted, Java is an extremely dynamic language. At runtime, the Java environment can extend itself by linking in classes that may be located on remote servers on a network (for example, the Internet)

At runtime, the Java interpreter performs name resolution while linking in the necessary classes. The Java interpreter is also responsible for determining the placement of objects in memory. These two features of the Java interpreter solve the problem of changing the definition of a class used by other classes

## **Java Virtual Machine**

When Java was created, the goal was to create a machine-independent programming language that then could be compiled into a portable binary format. In theory, that is exactly what was achieved. Java code is portable to any system that has a Java interpreter. However, Java is not at all machine independent. Rather, Java is machine specific to the Java virtual machine

## **Language Fundamental**

### **First Program**

1. Start NotePad. In a new document, type in the following code:

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

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

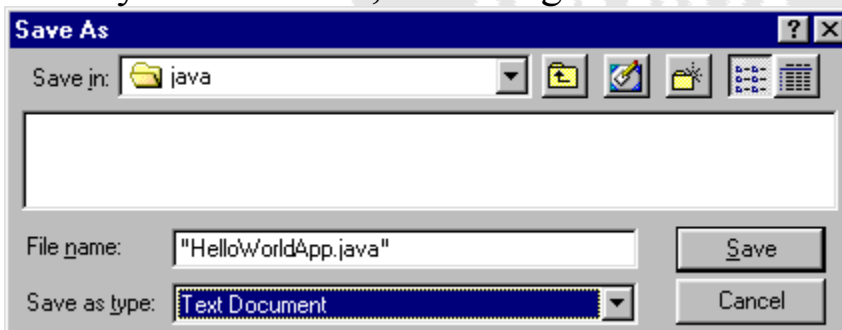
2. Save this code to a file. From the menu bar, select File > Save As. In the Save As dialog box:

Using the Save in drop-down menu, specify the folder (directory) where you'll save your file. In this example, the directory is **java** on the **C** drive.

In the File name text box, type "**HelloWorldApp.java**", including the double quotation marks.

From the Save as type drop-down menu, choose Text Document.

When you're finished, the dialog box should look like this:

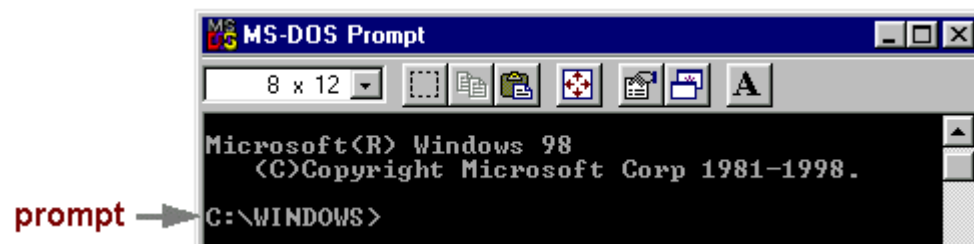


Now click Save, and exit NotePad.

## Compiling a program

### Compile the Source File.

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



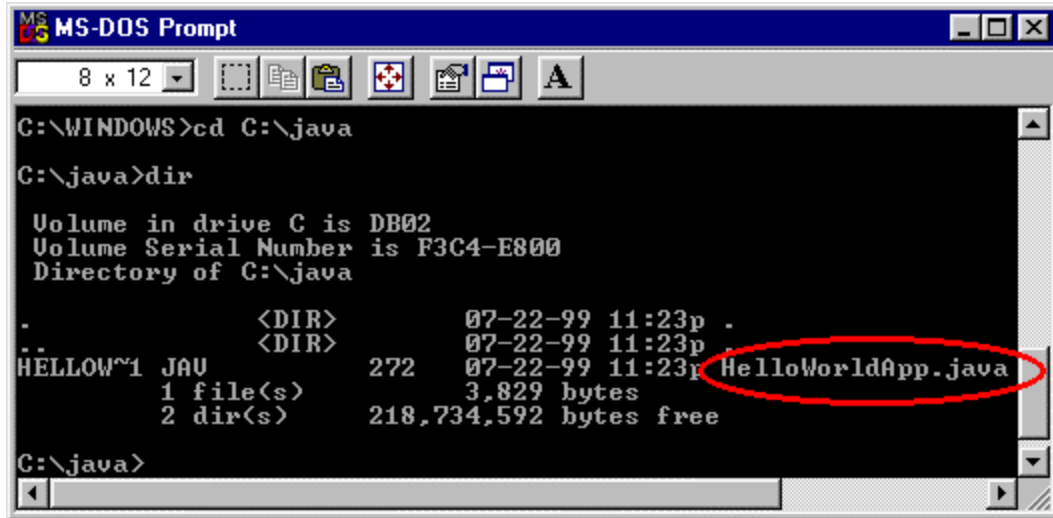
The prompt shows your *current directory*. When you bring up the prompt for Windows 95/98, your current directory is usually WINDOWS on your C drive (as shown above) or WINNT for Windows NT. To compile your source code file, change your current directory to the directory where your file is located. For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

```
cd c:\java
```

Now the prompt should change to C:\java>.

If you enter dir at the prompt, you should see your file.



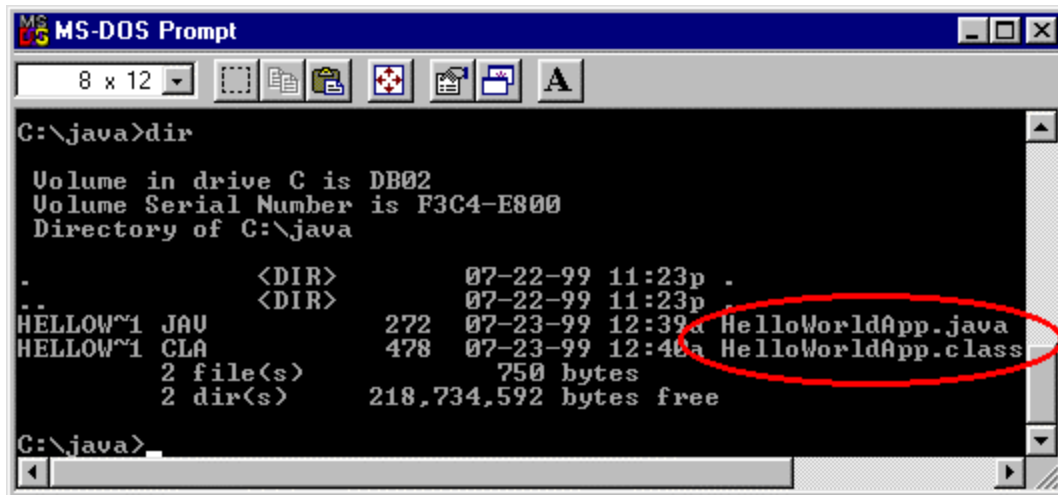


```
MS-DOS Prompt
8 x 12
C:\WINDOWS>cd C:\java
C:\java>dir
Volume in drive C is DB02
Volume Serial Number is F3C4-E800
Directory of C:\java
.                <DIR>          07-22-99 11:23p .
..               <DIR>          07-22-99 11:23p ..
HELLOW~1 JAU    272      07-22-99 11:23p HelloWorldApp.java
1 file(s)        3,829 bytes
2 dir(s)         218,734,592 bytes free
C:\java>
```

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

```
javac
HelloWorldApp.java
```

The compiler has generated a Java bytecode file, HelloWorldApp.class. At the prompt, type dir to see the new file that was generated:



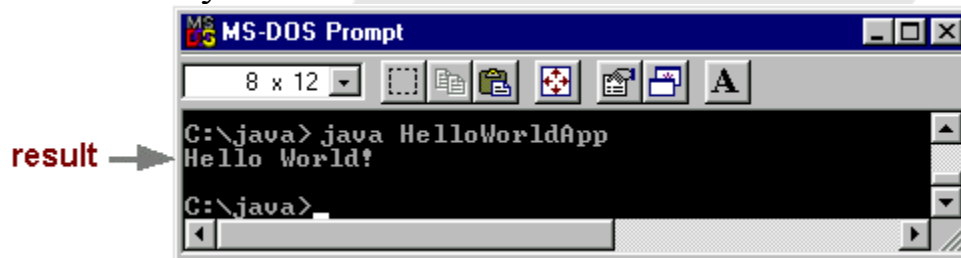
Now that you have a .class file, you can run your program.

### Running a program

In the same directory, enter at the prompt:

```
java HelloWorldApp
```

Now you should see:



### Command line argument

All Java Application contain a static method named `main()`. This method takes one argument that is an array of String objects.. These objects represent any arguments that may have been entered by the user on the command line.

## Accepting user inputs

Class CommandLineArgs

```
{
    public static void main(String args[])
    {
        System.out.println("args.length =" + args.length);
        System.out.println("args[0] =" + args[0]);
        System.out.println("args[1] =" + args[1]);
        System.out.println("args[2] =" + args[2]);
    }
}
```

You may invoke as follows:

```
Java CommandLineArgs 1 2 abc
```

Output will be,

```
Args.length=3
Args[0]=1
Args[1]=2
Args[2]=abc
```

## Datatypes

You may remember two data types already, these being floating point (represented by the `float` keyword) and integer (represented by the `int` keyword). Java has eight different data types, all of which represent different kinds of values in a program. These data types are `byte`, `short`, `int`, `long`, `float`, `double`, `char`, and `boolean`. In this section, you'll learn what kinds of values these various data types represent

## Integer Types

The most common values used in computer programs are integers, which represent whole number values such as 12, 1988, and -34. Integer

values can be both positive or negative, or even the value 0. The size of the value that's allowed depends on the integer data type you choose.

Java features four integer data types, which are **byte**, **short**, **int**, and **long**.

## **Byte**

The first integer type, **byte**, takes up the least amount of space in a computer's memory. When you declare a constant or variable as **byte**, you are limited to values in the range -128 to 127. Why would you want to limit the size of a value in this way? Because the smaller the data type, the faster the computer can manipulate it. For example, your computer can move a **byte** value, which consumes only eight bits of memory, much faster than an **int** value, which, in Java, is four times as large.

In Java, you declare a **byte** value like this:

```
byte identifier;
```

In the preceding line, **byte** is the data type for the value, and **identifier** is the variable's name. You can also simultaneously declare and assign a value to a variable like this:

```
byte count = 100;
```

After Java executes the preceding line, your program will have a variable named **count** that currently holds the value of 100. Of course, you can change the contents of **count** at any time in your program.

## Short

The next biggest type of Java integer is short. A variable declared as short can hold a value from -32,768 to 32,767. You declare a short value like this:

```
short identifier;
```

or

```
short identifier = value;
```

In the preceding line, value can be any value from -32,768 to 32,767, as described previously. In Java, short values are twice as big in memory- 16 bits (or two bytes)-as byte values.

## Int

Next in the integer data types is int, which can hold a value from -2,147,483,648 to 2,147,483,647. Now you're getting into some big numbers! The int data type can hold such large numbers because it takes up 32 bits (four bytes) of computer memory. You declare int values like this:

```
int identifier;
```

or

```
int identifier = value;
```

## Long

The final integer data type in the Java language is long, which takes up a whopping 64 bits (eight bytes) of computer memory and can hold truly immense numbers. Unless you're calculating the number of molecules in the universe, you don't even have to know how big a long number can be. I'd figure it out for you, but I've never seen a calculator that can handle numbers that big. You declare a long value like this:

```
long identifier;
```

or

```
long identifier = value;
```

## **Floating-Point Types**

Whereas integer values can hold only whole numbers, the floating-point data types can hold values with both whole number and fractional parts. Examples of floating-point values include 32.9, 123.284, and -43.436. As you can see, just like integers, floating-point values can be either positive or negative.

Java includes two floating-point types, which are float and double.

### **Float**

In Java, a value declared as float can hold a number in the range from around  $-3.402823 \times 10^{38}$  to around  $3.402823 \times 10^{38}$ . These types of values are also known as single-precision floating-point numbers and take up 32 bits (four bytes) of memory. You declare a single-precision floating-point number like this:

```
float identifier;
```

or

```
float identifier = value;
```

## **double**

The second type of floating-point data, double, represents a double-precision value, which is a much more accurate representation of floating-point numbers because it allows for more decimal places. A double value can be in the range from  $-1.79769313486232 \times 10^{308}$  to  $1.79769313486232 \times 10^{308}$  and is declared like this:

```
double identifier;
```

or

```
double identifier = value;
```

## **Characters**

Often in your programs, you'll need a way to represent character values rather than just numbers. A character is a symbol that's used in text. The most obvious examples of characters are the letters of the alphabet, in both upper- and lowercase varieties. There are, however, many other characters, including not only things such as spaces, exclamation points, and commas, but also tabs, carriage returns, and line feeds. The symbols 0 through 9 are also characters when they're not being used in mathematical calculations.

In order to provide storage for character values, Java features the char data type, which is 16 bits. However, the size of the char data type has little to do with the values it can hold. Basically, you can think of a char as being able to hold a single character. (The 16 bit length accommodates Unicode characters, which you don't need to worry about in this book.) You declare a char value like this:

```
char c;
```

or

```
char c = 'A';
```

## Boolean

Many times in a program, you need a way to determine if a specific condition has been met. For example, you might need to know whether a part of your program executed properly. In such cases, you can use Boolean values, which are represented in Java by the boolean data type. Boolean values are unique in that they can be only one of two possible values: true or false. You declare a boolean value like this:

```
boolean identifier;
```

or

```
boolean identifier = value;
```

In the second example, value must be true or false. In an actual program, you might write something like this:

```
boolean file_okay = true;
```

Boolean values are often used in if statements, which enable you to do different things depending on the value of a variable.

## Literals

### Special Character Literals.

<i>Character</i>	<i>Symbol</i>
Backslash	\\
Backspace	\b



Carriage return	\r
Double quote	\"
Form feed	\f
Line feed	\n
Single quote	\'
Tab	\t

## Variables

Variables are values that can change as much as needed during the execution of a program. Because of a variable's changing nature, there's no such thing as a hard-coded variable. That is, hard-coded values in a program are always constants (or, more accurately, literals).

**Definition:** A variable is an item of data named by an identifier.

## Declaration

You must explicitly provide a name and a type for each variable you want to use in your program. The variable's name must be a legal *identifier* --an unlimited series of Unicode characters that begins with a letter. You use the variable name to refer to the data that the variable contains. The variable's type determines what values it can hold and what operations can be performed on it. To give a variable a type and a name, you write a variable declaration, which generally looks like this:

*type name;*

## Dynamic Initialization

The MaxVariablesDemo program, shown below, declares eight variables of different types within its main method. The variable declarations are red:

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

        // integers
        byte largestByte = Byte.MAX_VALUE;
        short largestShort = Short.MAX_VALUE;
        int largestInteger =
Integer.MAX_VALUE;
        long largestLong = Long.MAX_VALUE;

        // real numbers
        float largestFloat = Float.MAX_VALUE;
        double largestDouble =
Double.MAX_VALUE;

        // other primitive types
        char aChar = 'S';
        boolean aBoolean = true;
```

```
// display them all
System.out.println("The largest byte
value is " + largestByte);
System.out.println("The largest short
value is " + largestShort);
System.out.println("The largest integer
value is " + largestInteger);
System.out.println("The largest long
value is " + largestLong);

System.out.println("The largest float
value is " + largestFloat);
System.out.println("The largest double
value is " + largestDouble);

if (Character.isUpperCase(aChar)) {
    System.out.println("The character " +
aChar + " is upper case.");
} else {
    System.out.println("The character " +
aChar + " is lower case.");
}
```

```
        System.out.println("The value of  
aBoolean is " + aBoolean);  
    }  
}
```

The output from this program is:

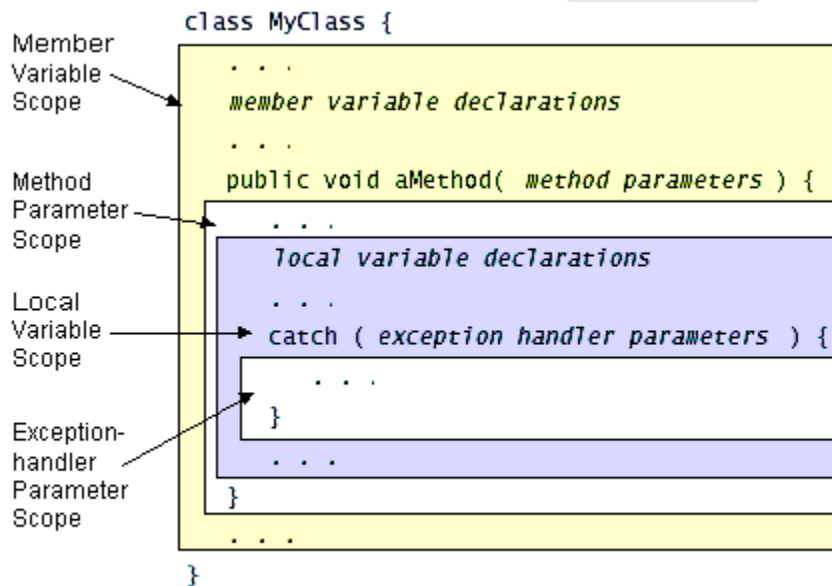
```
The largest byte value is 127  
The largest short value is 32767  
The largest integer value is 2147483647  
The largest long value is  
9223372036854775807  
The largest float value is 3.40282e+38  
The largest double value is 1.79769e+308  
The character S is upper case.  
The value of aBoolean is true
```

## Scope

A variable's scope is the region of a program within which the variable can be referred to by its simple name. Secondly, scope also determines when the system creates and destroys memory for the variable. Scope is distinct from visibility, which applies only to member variables and determines whether the variable can be used from outside of the class within which it is declared. Visibility is set with an access modifier.

The location of the variable declaration within your program establishes its scope and places it into one of these four categories:

- *member variable*
- *local variable*
- method parameter
- exception-handler parameter



A member variable is a member of a class or an object. It is declared within a class but outside of any method or constructor. A member variable's scope is the entire declaration of the class.

You declare local variables within a block of code. In general, the scope of a local variable extends from its declaration to the end of the code block in which it was declared. In `MaxVariablesDemo`, all of the variables declared within the `main` method are local variables.

Parameters are formal arguments to methods or constructors and are used to pass values into methods and constructors. The scope of a parameter is the entire method or constructor for which it is a parameter.

Exception-handler parameters are similar to parameters but are arguments to an exception handler rather than to a method or a constructor. The scope of an exception-handler parameter is the code block between { and } that follow a catch statement. Consider the following code sample:

```
if (...) {  
    int i = 17;  
    ...  
}  
System.out.println("The value of i = " + i); // error
```

### **Automatic Type Promotion in Expression**

```
class TypePromotion {  
    public static void main(String args[]) {  
        int i;  
        float f;  
        i = 10;  
        f = 23.25f;  
        System.out.println(i * f);  
    }  
}
```

## Operators

An operator performs a function on one, two, or three operands. An operator that requires one operand is called a *unary operator*. For example, `++` is a unary operator that increments the value of its operand by 1. An operator that requires two operands is a Binary Operator. For example, `=` is a binary operator that assigns the value from its right-hand operand to its left-hand operand. And finally, a *ternary operator* is one that requires three operands. The Java programming language has one ternary operator, `?:`, which is a short-hand `if-else` statement.

## Arithmetic

The Java programming language supports various arithmetic operators for all floating-point and integer numbers. These operators are `+` (addition), `-` (subtraction), `*` (multiplication), `/` (division), and `%` (modulo).

The following table summarizes the binary arithmetic operations in the Java programming language.

Operator	Use	Description
<code>+</code>	<code>op1 + op2</code>	Adds op1 and op2
<code>-</code>	<code>op1 - op2</code>	Subtracts op2 from op1
<code>*</code>	<code>op1 * op2</code>	Multiplies op1 by op2
<code>/</code>	<code>op1 / op2</code>	Divides op1 by op2
<code>%</code>	<code>op1 % op2</code>	Computes the remainder of dividing op1 by op2

When an integer and a floating-point number are used as operands to a single arithmetic operation, the result is floating point. The integer is implicitly converted to a floating-point number before the operation takes place.

In addition to the binary forms of + and -, each of these operators has unary versions that perform the following operations: Operator	Use	Description
+	+op	Promotes op to int if it's a byte, short, or char
-	-op	Arithmetically negates op

```

public class Arithmetic {
    public static void main(String args[]) {
        System.out.print(5/2);
        System.out.print(" " + 5%2);
        System.out.print(" " + 4/2);
        System.out.println(" " + 4%2);
    }
}

```



out put is ::

2 1 2 0

## Relational and Logical

A relational operator compares two values and determines the relationship between them. For example, `!=` returns true if the two operands are unequal. This table summarizes the relational operators:

Operator	Use	Returns true if
<code>&gt;</code>	<code>op1 &gt; op2</code>	op1 is greater than op2
<code>&gt;=</code>	<code>op1 &gt;= op2</code>	op1 is greater than or equal to op2
<code>&lt;</code>	<code>op1 &lt; op2</code>	op1 is less than op2
<code>&lt;=</code>	<code>op1 &lt;= op2</code>	op1 is less than or equal to op2
<code>==</code>	<code>op1 == op2</code>	op1 and op2 are equal
<code>!=</code>	<code>op1 != op2</code>	op1 and op2 are not equal

## Assignment

You use the basic assignment operator, `=`, to assign one value to another.

```
byte largestByte = Byte.MAX_VALUE;
short largestShort = Short.MAX_VALUE;
int largestInteger = Integer.MAX_VALUE;
long largestLong = Long.MAX_VALUE;
```

The Java programming language also provides several shortcut assignment operators that allow you to perform an arithmetic, shift, or bitwise operation and an assignment operation all with one operator. Suppose you wanted to add a number to a variable and assign the result back into the variable, like this:

```
i = i + 2;
```

You can shorten this statement using the shortcut operator `+=`, like this:

```
i += 2;
```

The two previous lines of code are equivalent.

The following table lists the shortcut assignment operators and their lengthy equivalents

Operator	Use	Equivalent to
<code>+=</code>	<code>op1 += op2</code>	<code>op1 = op1 + op2</code>
<code>-=</code>	<code>op1 -= op2</code>	<code>op1 = op1 - op2</code>
<code>*=</code>	<code>op1 *= op2</code>	<code>op1 = op1 * op2</code>
<code>/=</code>	<code>op1 /= op2</code>	<code>op1 = op1 / op2</code>
<code>%=</code>	<code>op1 %= op2</code>	<code>op1 = op1 % op2</code>
<code>&amp;=</code>	<code>op1 &amp;= op2</code>	<code>op1 = op1 &amp; op2</code>
<code> =</code>	<code>op1  = op2</code>	<code>op1 = op1   op2</code>
<code>^=</code>	<code>op1 ^= op2</code>	<code>op1 = op1 ^ op2</code>

<<=	op1 <<= op2	op1 = op1 << op2
>>=	op1 >>= op2	op1 = op1 >> op2
>>>=	op1 >>>= op2	op1 = op1 >>> op2

## Conditional / Ternary

Relational operators often are used with conditional operators to construct more complex decision-making expressions. The Java programming language supports six conditional operators-five binary and one unary--as shown in the following table.

Operator	Use	Returns true if
&&	op1 && op2	op1 and op2 are both true, conditionally evaluates op2
	op1    op2	either op1 or op2 is true, conditionally evaluates op2
!	! op	op is false
&	op1 & op2	op1 and op2 are both true, always evaluates op1 and op2
	op1   op2	either op1 or op2 is true, always evaluates op1 and op2
^	op1 ^ op2	if op1 and op2 are different--that is if one or the other of the operands is true but not both

One such operator is &&, which performs the conditional AND operation. You can use two different relational operators along with &&

to determine whether both relationships are true. The following line of code uses this technique to determine whether an array index is between two boundaries. It determines whether the index is both greater than or equal to 0 and less than NUM\_ENTRIES, which is a previously defined constant value.

```
0 <= index && index < NUM_ENTRIES
```

The && operator will return true only if both operands are true.

When both operands are boolean, the operator & performs the same operation as &&. However, & always evaluates both of its operands and returns true if both are true. Likewise, when the operands are boolean, | performs the same operation as ||. The | operator always evaluates both of its operands and returns true if at least one of its operands is true. When their operands are numbers, & and | perform bitwise manipulations.

The ?: operator is a conditional operator that is short-hand for an if-else statement:

```
op1 ? op2 : op3
```

The ?: operator returns op2 if op1 is true or returns op3 if op1 is false.

## Increment – Decrement

The shortcut increment/decrement operators are summarized in the following table.

Operator	Use	Description
++	op++	Increments on by 1. evaluates to the value of on

		before it was incremented
++	++op	Increments op by 1; evaluates to the value of op after it was incremented
--	op--	Decrements op by 1; evaluates to the value of op before it was decremented
--	--op	Decrements op by 1; evaluates to the value of op after it was decremented

## Bitwise Shift

A shift operator performs bit manipulation on data by shifting the bits of its first operand right or left. This table summarizes the shift operators available in the Java programming language.

Operator	Use	Operation
>>	op1 >> op2	shift bits of op1 right by distance op2
<<	op1 << op2	shift bits of op1 left by distance op2
>>>	op1 >>> op2	shift bits of op1 right by distance op2 (unsigned)

```
class Shift {
    public static void main (String args[]) {
        int x = 7;
        System.out.println("x = " + x);
    }
}
```

```
System.out.println("x >> 2 = " + (x >> 2));  
System.out.println("x << 1 = " + (x << 1));  
System.out.println("x >>> 1 = " + (x >>> 1));  
}  
  
}
```

The output of Shift follows:

```
x = 7  
x >> 2 = 1  
x << 1 = 14  
x >>> 1 = 3
```

## Operator Precedence

Even though Java expressions are typically evaluated from left to right, there still are many times when the result of an expression would be indeterminate without other rules. The following expression illustrates the problem:

```
x = 2 * 6 + 16 / 4
```

Strictly using the left-to-right evaluation of the expression, the multiplication operation  $2 * 6$  is carried out first, which leaves a result of 12. The addition operation  $12 + 16$  is then performed, which gives a result of 28. The division operation  $28 / 4$  is then performed, which gives

a result of 7. Finally, the assignment operation  $x = 7$  is handled, in which the number 7 is assigned to the variable x.

Table ::

.	[]	()	
++	--	!	~
*	/	%	
+	-		
<<	>>	>>>	
<	>	<=	>=
==	!=		
&			
^			
&&			
?:			
=			

Evaluation of expressions still moves from left to right, but only when

dealing with operators that have the same precedence. Otherwise, operators with a higher precedence are evaluated before operators with a lower precedence.

## Control Statements

Statement Type	Keyword
looping	while, do-while , for
decision making	if-else, switch-case
exception handling	try-catch-finally, throw
branching	break, continue, label:, return

Programs use control flow statements to conditionally execute statements, to loop over statements, or to jump to another area in the program.

### If ... else

The If statement enables your program to selectively execute other statements, based on some criteria.

This is the simplest version of the if statement: The block of statements will be executed if a condition is true. Generally, the simple form of if can be written like this:

```
if (expression) {
    statement(s)
}
```

What if you want to perform a different set of statements if the expression is false? You use the else.

The else block is executed if the if part is false. Another form of the else statement, else if, executes a statement based on another expression. An



if statement can have any number of companion else if statements but only one else. Following is a program, IfElseDemo♦, that assigns a grade based on the value of a test score: an A for a score of 90% or above, a B for a score of 80% or above, and so on:

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

        int testscore = 76;
        char grade;

        if (testscore >= 90) {
            grade = 'A';
        } else if (testscore >= 80) {
            grade = 'B';
        } else if (testscore >= 70) {
            grade = 'C';
        } else if (testscore >= 60) {
            grade = 'D';
        } else {
            grade = 'F';
        }
        System.out.println("Grade = " + grade);
    }
}
```

Out Put ::  
Grade = c

### Switch ... case

Use the *switch* statement to conditionally perform statements based on an integer expression. Following is a sample program, SwitchDemo, that declares an integer named month whose value supposedly

represents the month in a date. The program displays the name of the month, based on the value of month, using the switch statement:

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

        int month = 8;
        switch (month) {
            case 1: System.out.println("January"); break;
            case 2: System.out.println("February"); break;
            case 3: System.out.println("March"); break;
            case 4: System.out.println("April"); break;
            case 5: System.out.println("May"); break;
            case 6: System.out.println("June"); break;
            case 7: System.out.println("July"); break;
            case 8: System.out.println("August"); break;
            case 9: System.out.println("September"); break;
            case 10: System.out.println("October"); break;
            case 11: System.out.println("November"); break;
            case 12: System.out.println("December"); break;
            default: System.out.println("Hey, that's not a valid month!");
        }
        break;
    }
}

Out Put ::August
```

## Loops

Loops enable you to execute code repeatedly. There are three types of loops in Java: for loops, while loops, and do-while loops.

## While

You use a *while* statement to continually execute a block of statements while a condition remains true. The general syntax of the **while** statement is:

```
while (expression) {
    statement
}
```

First, the **while** statement evaluates *expression*, which must return a boolean value. If the expression returns true, then the **while** statement executes the statement(s) associated with it. The **while** statement continues testing the expression and executing its block until the expression returns **false**.

The example program shown below, called WhileDemo, uses a while statement to step through the characters of a string, appending each character from the string to the end of a string buffer until it encounters the letter g.

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

        String copyFromMe = "Copy this string until you " +
            "encounter the letter 'g'.";
        StringBuffer copyToMe = new StringBuffer();

        int i = 0;
        char c = copyFromMe.charAt(i);

        while (c != 'g') {
            copyToMe.append(c);
```

```

        c = copyFromMe.charAt(++i);
    }
    System.out.println(copyToMe);
}
}

```

The value printed by the last line is: Copy this string.

### do ... while

The Java programming language provides another statement that is similar to the while statement--the *do-while* statement. The general syntax of the do-while is:

```

do {
    statement(s)
} while (expression);

```

Instead of evaluating the expression at the top of the loop, do-while evaluates the expression at the bottom. Thus the statements associated with a do-while are executed at least once.

Here's the previous program rewritten to use do-while and renamed to

```

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

        String copyFromMe = "Copy this string until you " +
            "encounter the letter 'g'.";
        StringBuffer copyToMe = new StringBuffer();

        int i = 0;
        char c = copyFromMe.charAt(i);

        do {
            copyToMe.append(c);
            c = copyFromMe.charAt(++i);
        } while (c != 'g');
        System.out.println(copyToMe);
    }
}

```

The value printed by the last line is: Copy this string.

## For

The *for* statement provides a compact way to iterate over a range of values. The general form of the for statement can be expressed like this:

```
for (initialization; termination; increment) {
    statement
}
```

The *initialization* is an expression that initializes the loop-it's executed once at the beginning of the loop. The *termination* expression determines when to terminate the loop. This expression is evaluated at the top of each iteration of the loop. When the expression evaluates to false, the loop terminates. Finally, *increment* is an expression that gets invoked after each iteration through the loop. All these components are optional. In fact, to write an infinite loop, you omit all three expressions:

```
for ( ; ; ) { // infinite loop
    ...
}
```

Often for loops are used to iterate over the elements in an array, or the characters in a string. The following sample, ForDemo♦, uses a for statement to iterate over the elements of an array and print them:

```
public class ForDemo {
    public static void main(String[] args) {
        int[] arrayOfInts = { 32, 87, 3, 589, 12, 1076,
                               2000, 8, 622, 127 };

        for (int i = 0; i < arrayOfInts.length; i++) {
            System.out.print(arrayOfInts[i] + " ");
        }
        System.out.println();
    }
}
```

The output of the program is: 32 87 3 589 12 1076 2000 8 622 127.

## Nested Loops and Labeled Loops

A *label* is an identifier placed before a statement. The label is followed by a colon (:):

*statementName: someJavaStatement;*

. The following program, `BreakWithLabelDemo`, searches for a value in a two-dimensional array. Two nested for loops traverse the array. When the value is found, a labeled break terminates the statement labeled search, which is the outer for loop:

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

        int[][] arrayOfInts = { { 32, 87, 3, 589 },
                                { 12, 1076, 2000, 8 },
                                { 622, 127, 77, 955 }
                                };
        int searchfor = 12;

        int i = 0;
        int j = 0;
        boolean foundIt = false;

        search:
        for ( ; i < arrayOfInts.length; i++) {
            for (j = 0; j < arrayOfInts[i].length; j++) {
                if (arrayOfInts[i][j] == searchfor) {
                    foundIt = true;
                    break search;
                }
            }
        }
    }
}
```

```

    if (foundIt) {
        System.out.println("Found " + searchfor + " at " + i + ", " + j);
    } else {
        System.out.println(searchfor + "not in the array");
    }
}
}
}

```

The output of this program is:

```
Found 12 at 1, 0
```

The labeled form of the continue statement skips the current iteration of an outer loop marked with the given label. The following example program, `ContinueWithLabelDemo`, uses nested loops to search for a substring within another string. Two nested loops are required: one to iterate over the substring and one to iterate over the string being searched. This program uses the labeled form of continue to skip an iteration in the outer loop:

```

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

        String searchMe = "Look for a substring in me";
        String substring = "sub";
        boolean foundIt = false;

        int max = searchMe.length() - substring.length();

    test:
        for (int i = 0; i <= max; i++) {
            int n = substring.length();
            int j = i;
            int k = 0;
            while (n-- != 0) {
                if (searchMe.charAt(j++) != substring.charAt(k++)) {

```

```

        continue test;
    }
}
foundIt = true;
    break test;
}
System.out.println(foundIt ? "Found it" : "Didn't find it");
}
}

```

Here is the output from this program:

Found it

## Jump Statements

### Break

The *break* statement has two forms: unlabeled and labeled. You saw the unlabeled form of the **break** statement used with **switch** earlier. As noted there, an unlabeled **break** terminates the enclosing **switch** statement, and flow of control transfers to the statement immediately following the **switch**. You can also use the unlabeled form of the **break** statement to terminate a **for**, **while**, or **do-while** loop. The following sample program, **BreakDemo**, contains a **for** loop that searches for a particular value within an array:

```

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

        int[] arrayOfInts = { 32, 87, 3, 589, 12,
1076,

```



```
                2000, 8, 622, 127 };
```

```
int searchfor = 12;
```

```
int i = 0;
```

```
boolean foundIt = false;
```

```
for ( ; i < arrayOfInts.length; i++) {
```

```
    if (arrayOfInts[i] == searchfor) {
```

```
        foundIt = true;
```

```
        break;
```

```
    }
```

```
}
```

```
if (foundIt) {
```

```
    System.out.println("Found " +
```

```
searchfor + " at index " + i);
```

```
    } else {
```

```
        System.out.println(searchfor + "not in
```

```
the array");
```

```
    }
```

```
}
```

```
}
```

The **break** statement terminates the **for** loop when the value is found. The flow of control transfers to the statement following the enclosing **for**, which is the print statement at the end of the program.

The output of this program is:

Found 12 at index 4

### **Continue**

You use the *continue* statement to skip the current iteration of a **for**, **while**, or **do-while** loop. The unlabeled form skips to the end of the innermost loop's body and evaluates the boolean expression that controls the loop, basically skipping the remainder of this iteration of the loop. The following program, `ContinueDemo`, steps through a string buffer checking each letter. If the current character is not a **p**, the `continue` statement skips the rest of the loop and proceeds to the next character. If it is a **p**, the program increments a counter, and converts the **p** to an uppercase letter.

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

        StringBuffer searchMe = new StringBuffer(
            "peter piper picked a peck of pickled peppers");
        int max = searchMe.length();
        int numPs = 0;

        for (int i = 0; i < max; i++) {
            //interested only in p's
            if (searchMe.charAt(i) != 'p')
                continue;
        }
    }
}
```

```
        //process p's
        numPs++;
        searchMe.setCharAt(i, 'P');
    }
    System.out.println("Found " + numPs + " p's in the string.");
    System.out.println(searchMe);
}
}
```

Here is the output of this program:

Found 9 p's in the string.

Peter PiPer Picked a Peck of Pickled PePPers

## Return

The last of Java's branching statements is the *return* statement. You use *return* to exit from the current method. The flow of control returns to the statement that follows the original method call. The *return* statement has two forms: one that returns a value and one that doesn't. To return a value, simply put the value (or an expression that calculates the value) after the *return* keyword:

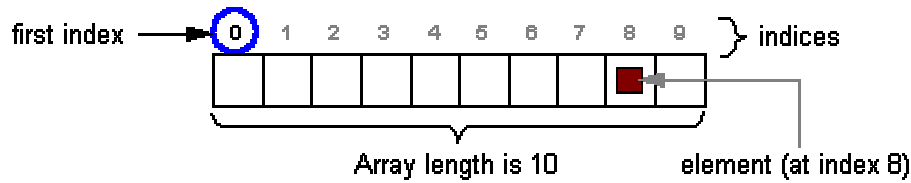
```
return ++count;
```

The data type of the value returned by *return* must match the type of the method's declared return value. When a method is declared void, use the form of *return* that doesn't return a value:

```
return;
```

## Arrays

An array is a structure that holds multiple values of the same type. The length of an array is established when the array is created (at runtime). After creation, an array is a fixed-length structure.



An *array element* is one of the values within an array and is accessed by its position within the array.

If you want to store data of different types in a single structure, or if you need a structure

*Arrays* are objects that contain a number of variables of the same type. These component variables are referenced using the integer indices  $0, \dots, n-1$ , where  $n$  is the length of the array. The type of the array is identified by appending `[]` to the type of its components. For example, `int[]` identifies an array of type `int`, `Object[]` identifies an array of type `Object`, and `char[][]` identifies an array of an array of type `char`.

### Array Allocation

When a variable of an array type is declared, the size of the array is not identified, and the array object is not allocated. To allocate storage for an array, you can use the `new` operator to create an array object of a specific size. For example, the following statement:  
`char ch[] = new char[24];`

creates a char array of length 24, the individual component variables of which can be referenced by `ch[0]`, `ch[2]`, ..., `ch[23]`. The following statement creates an array of type `Dice[]` of length 6:

```
Dice[] d = new Dice[6];
```

Arrays can also be allocated by specifying their initial values. For example, the following allocates a `String` array of length 7 that contains abbreviations for the days of the week:

```
String days[] = {"sun", "mon", "tue", "wed", "thu", "fri", "sat"};
```

The length of an array can always be found by appending `.length` to the name of the array. For example, `days.length` returns the integer 7 as the length of `days[]`.

### Alternate Array Declaration Syntax

Arrays are declared by declaring a variable to be of an array type. For example, the following declares `nums` to be an array of type `int`:

```
int[] nums;
```

The declaration can also be written as follows:

```
int nums[];
```

You can place the brackets after either the type or the variable name.

### One-Dimensional Array

Here's a simple program, called `ArrayDemo`, that creates the array, puts some values in it, and displays the values.

```
public class ArrayDemo {
    public static void main(String[] args) {
        int[] anArray;          // declare an array of integers

        anArray = new int[10];  // create an array of integers

        // assign a value to each array element and print
        for (int i = 0; i < anArray.length; i++) {
            anArray[i] = i;
        }
    }
}
```

```

        System.out.print(anArray[i] + " ");
    }
    System.out.println();
}
}

```

## Multi-Dimensional Array

As with Multi-dimensional Array ,you must explicitly create the sub-arrays within an array. So if you don't use an initializer, you need to write code like the following, which you can find in:

ArrayOfArraysDemo2

```

public class ArrayOfArraysDemo2 {
    public static void main(String[] args) {
        int[][] aMatrix = new int[4][];

        //populate matrix
        for (int i = 0; i < aMatrix.length; i++) {
            aMatrix[i] = new int[5]; //create sub-array
            for (int j = 0; j < aMatrix[i].length; j++) {
                aMatrix[i][j] = i + j;
            }
        }

        //print matrix
        for (int i = 0; i < aMatrix.length; i++) {
            for (int j = 0; j < aMatrix[i].length; j++) {
                System.out.print(aMatrix[i][j] + " ");
            }
            System.out.println();
        }
    }
}

```

You must specify the length of the primary array when you create the array. You can leave the length of the sub-arrays unspecified until you create them.

## **Implementing Classes**

### **Basics**

A class is a template or prototype that defines a type of object. A class is to an object what a blueprint is to a house. Many houses can be built from a single blueprint; the blueprint outlines the makeup of the houses. Classes work exactly the same way, except that they outline the makeup of objects.

Java, C++, Smalltalk, and some other object-oriented languages follow a class-based approach. This approach allows you to declare classes that serve as a template from which objects are created.

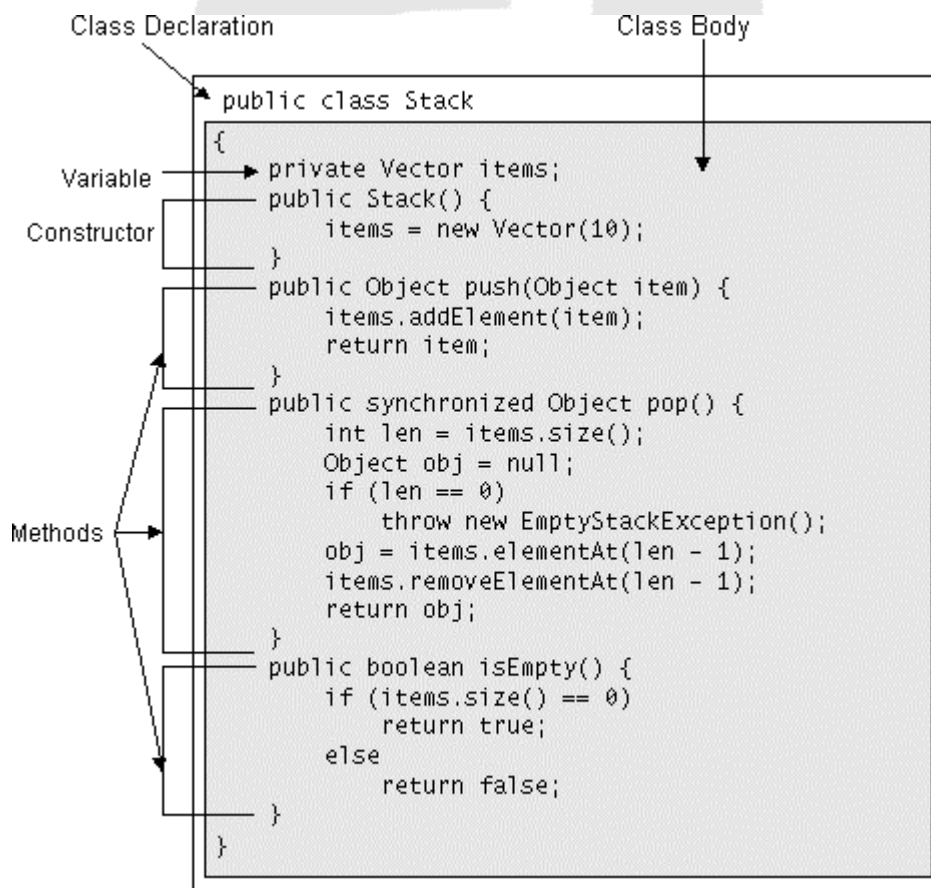
As you would expect, a class defines the type of data that is contained in an object and the methods that are used to access this data. A class also defines one or more methods to be used to create objects that are instances of the class. An instance of a class is a concrete manifestation of the class in your computer's memory.

For example, consider a job application form as an object. It contains data-the different form fields that must be filled out. There are also methods for accessing the data-for example, fill in form and read form. Now suppose that you develop an application form for a company that will use it for new job applicants. When a job is advertised, 100 potential applicants show up. In order for these applicants to use your form, they must all be given a unique instance of the form. These form instances are created by using the form you developed as a master copy and then duplicating the master copy as many times as needed to create each instance. The job applicants then fill in their instances of the form, using the fill in form method.

In the preceding example, the master form is analogous to a class. The master form defines the data to be contained in each of its instances and implicitly provides methods by which the data can be accessed. In the same way, a class defines the data that can be contained in an object as well as the methods that can be used to access this data.

## General Form of a Class

Now that we've covered how to create and use objects, and how objects are cleaned up, it's time to show you how to write the classes from which objects are created. This section shows you the main components of a class through a small example that implements a last-in-first-out (LIFO) stack. The following diagram lists the class and identifies the structure of the code.





## Creating Classes & their Objects

The left side of the following diagram shows the possible components of a class declaration in the order they should or must appear in your class declaration. The right side describes their purposes. The required components are the class keyword and the class name and are shown in bold. All the other components are optional, and each appears on a line by itself (thus "**extends** *Super*" is a single component). Italics indicates an identifier such as the name of a class or interface. If you do not explicitly declare the optional items, the Java compiler assumes certain defaults: a nonpublic, nonabstract, nonfinal subclass of Object that implements no interfaces.

<code>public</code>	Class is publicly accessible.
<code>abstract</code>	Class cannot be instantiated.
<code>final</code>	Class cannot be subclassed.
<b><code>cClass</code></b> <i>NameOfCClass</i>	<b>Name of the Class.</b>
<code>extends</code> <i>Super</i>	Superclass of the class.
<code>implements</code> <i>Interfaces</i>	Interfaces implemented by the class.
{ <i>ClassBody</i> }	

The following list provides a few more details about each class declaration component. It also provides references to sections later in this trail that talk about what each component means, how to use each, and how it affects your class, other classes, and your Java program.

### public

By default, a class can be used only by other classes in the same package. The public modifier declares that the class can be used by any class regardless of its package

### **abstract**

Declares that the class cannot be instantiated.

### **final**

Declares that the class cannot be subclassed.

### **class** *NameOfClass*

The **class** keyword indicates to the compiler that this is a class declaration and that the name of the class is *NameOfClass*.

### **extends** *Super*

The **extends** clause identifies *Super* as the superclass of the class, thereby inserting the class within the class hierarchy.

### **implements** *Interfaces*

To declare that your class implements one or more interfaces, use the keyword **implements** followed by a comma-separated list of the names of the interfaces implemented by the class.

### **class body**

contains all of the code that provides for the life cycle of the objects created from it: constructors for initializing new objects, declarations for the variables that provide the state of the class and its objects, methods to implement the behavior of the class and its objects, and in rare cases, a **finalize** method to provide for cleaning up an object after it has done its job.

Variables and methods collectively are called *members*.

**Note:** Constructors are not methods. Nor are they members.

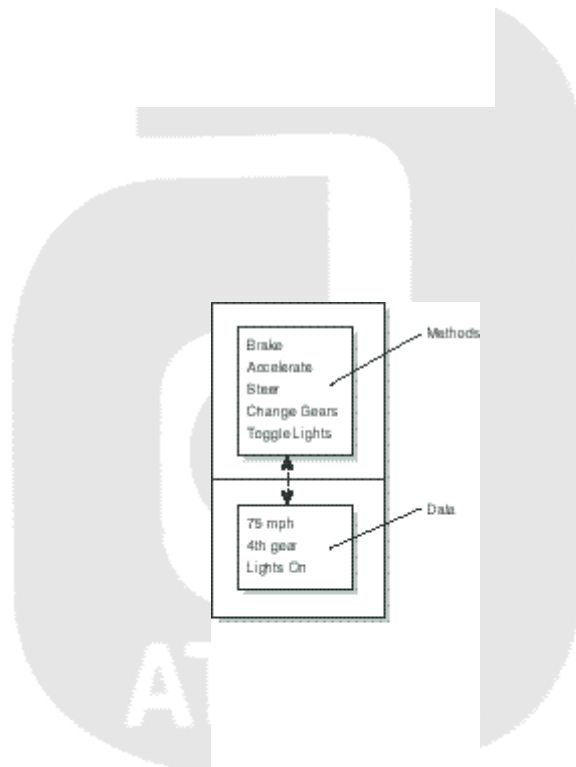
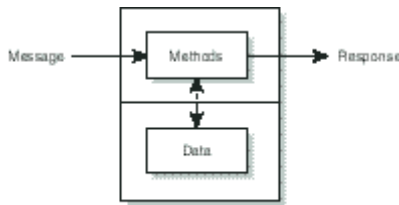
## Object

Objects are software bundles of data and the procedures that act on that data. The procedures are also known as methods. The merger of data and methods provides a means of more accurately representing real-world objects in software.

objects are at the heart of object-oriented technology. To understand how software objects are beneficial, think about the common characteristics of all real-world objects. Lions, cars, and calculators all share two common characteristics: state and behavior. For example, the state of a lion includes color, weight, and whether the lion is tired or hungry. Lions also have certain behaviors, such as roaring, sleeping, and hunting. The state of a car includes the current speed, the type of transmission, whether it is two-wheel or four-wheel drive, whether the lights are on, and the current gear, among other things. The behaviors for a car include turning, braking, and accelerating.

As with real-world objects, software objects also have these two common characteristics (state and behavior). To relate this back to programming terms, the state of an object is determined by its data; the behavior of an object is defined by its methods

Figure shows a visualization of a software object, including the primary components and how they relate. The software object in Figure clearly shows the two primary components of an object: data and methods. The figure also shows some type of communication, or access, between the data and the methods. Additionally, it shows how messages are sent through the methods, which result in responses from the object.



## Assigning Object Reference Variable

To create an instance of a class, you declare an object variable and use the new operator. When dealing with objects, a declaration merely states what type of object a variable is to represent. The object isn't actually created until the new operator is used. Following are two examples that use the new operator to create instances of the Alien class:

```
Alien anAlien = new Alien();
```

```
Alien anotherAlien;  
anotherAlien = new Alien(Color.red, 56, 24);
```

In the first example, the variable `anAlien` is declared and the object is created by using the `new` operator with an assignment directly in the declaration. In the second example, the variable `anotherAlien` is declared first; the object is created and assigned in a separate statement.

## Methods

Methods in Java determine the messages an object can receive.

The fundamental parts of a method are the name, the arguments, the return type, and the body. Here is the basic form:

```
returnType methodName( /* Argument list */ ) {  
    /* Method body */  
}
```

The return type is the type of the value that pops out of the method after you call it. The argument list gives the types and names for the information you want to pass into the method. The method name and argument list together uniquely identify the method.

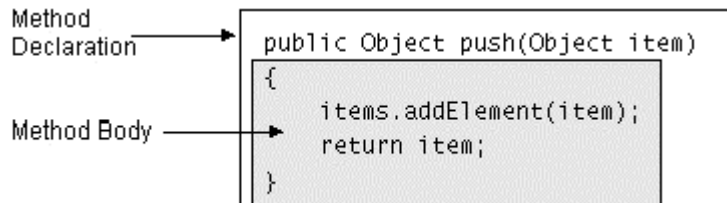
Methods in Java can be created only as part of a class. A method can be called only for an object, and that object must be able to perform that method call. If you try to call the wrong method for an object, you'll get an error message at compile-time. You call a method for an object by naming the object followed by a period (dot), followed by the name of the method and its argument list, like this:

`objectName.methodName(arg1, arg2, arg3)`. For example, suppose you have a method `f()` that takes no arguments and returns a value of type `int`. Then, if you have an object called `a` for which `f()` can be called, you can say this:

```
int x = a.f();
```

The type of the return value must be compatible with the type of `x`. This act of calling a method is commonly referred to as sending a message to an object. In the above example, the message is `f()` and the

object is a. Object-oriented programming is often summarized as simply “sending messages to objects.”



## Constructors

There is an important method you need to know about: the constructor. When you create an object, you typically want to initialize its member variables. The constructor is a special method you can implement in all your classes; it allows you to initialize variables and perform any other operations when an object is created from the class. The constructor is always given the same name as the class.

There is three types of constructor::  
Copy,parameterized,default

### The Alien class.

```
class Alien extends Enemy {  
    protected Color color;  
    protected int energy;  
    protected int aggression;
```

```
    public Alien() {  
        color = Color.green;  
        energy = 100;  
        aggression = 15;
```

```
}
```

```
public Alien(Color c, int e, int a) {  
    color = c;  
    energy = e;  
    aggression = a;  
}
```

```
public void move() {  
    // move the alien  
}
```

```
public void move(int x, int y) {  
    // move the alien to the position x,y  
}
```

```
public void morph() {  
    if (aggression < 10) {  
        // morph into a smaller size  
    }  
    else if (aggression < 20) {  
        // morph into a medium size  
    }  
}
```

```
    else {  
        // morph into a giant size  
    }  
}  
  
}
```

The Alien class uses method overloading to provide two different constructors. The first constructor is default Constructor which takes no parameters and initializes the member variables. The second constructor is parameterized constructor which takes the color, energy, and aggression of the alien and initializes the member variables with them.

### **The *this* keyword**

The *this* Keyword refers to the object that is currently executing .U will see that it is sometimes useful for a method to reference instance variables relative to the *this* keyword ,as follows :

This.varName

```
class Point3D {  
    double x;  
    double y;  
    double z;  
  
    Point3D(double x, double y, double z) {  
        this.x = x;  
        this.y = y;  
        this.z = z;  
    }  
}
```

```
class ThisKeywordDemo {
```



```
public static void main(String args[]) {  
    Point3D p = new Point3D(1.1, 3.4, -2.8);  
    System.out.println("p.x = " + p.x);  
    System.out.println("p.y = " + p.y);  
    System.out.println("p.z = " + p.z);  
}  
}
```

## Garbage Collection

When an object falls out of scope, it is removed from memory, or deleted. Similar to the constructor that is called when an object is created, Java provides the ability to define a destructor that is called when an object is deleted. Unlike the constructor, which takes on the name of the class, the destructor is called `finalize()`. The `finalize()` method provides a place to perform chores related to the cleanup of the object, and is defined as follows:

```
void finalize() {  
    // cleanup  
}
```

It is worth noting that the `finalize()` method is not guaranteed to be called by Java as soon as an object falls out of scope. The reason for this is that Java deletes objects as part of its system garbage collection, which occurs at inconsistent intervals. Because an object isn't actually deleted until Java performs a garbage collection, the `finalize()` method for the object isn't called until then either. Knowing this, it's safe to say that you shouldn't rely on the `finalize()` method for anything that is time critical. In general, you will rarely need to place code in the `finalize()` method simply because the Java runtime system does a pretty good job of cleaning up after objects on its own.

## Overloading

Another powerful object-oriented technique is method overloading. Method overloading enables you to specify different types of

information (parameters) to send to a method. To overload a method, you declare another version with the same name but different parameters.

For example, the `move()` method for the Alien class could have two different versions: one for general movement and one for moving to a specific location. The general version is the one you've already defined: it moves the alien based on its current state. The declaration for this version follows:

```
void move() {  
    // move the alien  
}
```

To enable the alien to move to a specific location, you overload the `move()` method with a version that takes `x` and `y` parameters, which specify the location to move to. The overloaded version of `move()` follows:

```
void move(int x, int y) {  
    // move the alien to position x,y  
}
```

Notice that the only difference between the two methods is the parameter lists; the first `move()` method takes no parameters; the second `move()` method takes two integers.

You may be wondering how the compiler knows which method is being called in a program, when they both have the same name. The compiler keeps up with the parameters for each method along with the name. When a call to a method is encountered in a program, the compiler checks the name and the parameters to determine which overloaded method is being called. In this case, calls to the `move()` methods are easily distinguishable by the absence or presence of the `int` parameters.

## Understanding final & static

The `final` modifier specifies that a variable has a constant value or that a method cannot be overridden in a subclass. To think of the `final` modifier

literally, it means that a class member is the final version allowed for the class.

Following are some examples of final member variables:

```
final public int numDollars = 25;
```

```
final boolean amIBroke = false;
```

final variables in Java are very similar to const variables in C++; they must always be initialized at declaration and their value can't change any time afterward.

The static modifier specifies that a variable or method is the same for all objects of a particular class.

Typically, new variables are allocated for each instance of a class. When a variable is declared as being static, it is allocated only once regardless of how many objects are instantiated. The result is that all instantiated objects share the same instance of the static variable. Similarly, a static method is one whose implementation is exactly the same for all objects of a particular class. This means that static methods have access only to static variables.

Following are some examples of a static member variable and a static method:

```
static int refCount;
```

```
static int getRefCount() {  
    return refCount;  
}
```

A beneficial side effect of static members is that they can be accessed without having to create an instance of a class. Remember the `System.out.println()` method. `out` is a static member variable of the `System` class, which means that you can access it without having to actually instantiate a `System` object.

## **Nested / Inner Classes**

Most Java classes are defined at the package level, meaning that each class is a member of a particular package. If you don't explicitly specify

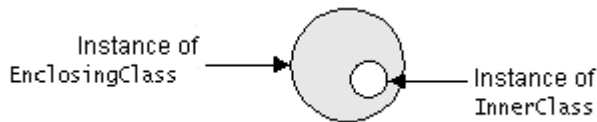
a package association for a class, the default package is assumed. Classes defined at the package level are known as top-level classes. Before Java 1.1, top-level classes were the only types of classes supported. However, Java 1.1 has ushered in a more open-minded approach to class definition. Java 1.1 supports inner classes, which are classes that can be defined in any scope. This means that a class can be defined as a member of another class, within a block of statements, or anonymously within an expression.

Rules governing the scope of an inner class closely match those governing variables. An inner class's name is not visible outside its scope, except in a fully qualified name (which helps in structuring classes within a package). The code for an inner class can use simple names from enclosing scopes--including class and member variables of enclosing classes--as well as local variables of enclosing blocks. In addition, you can define a top-level class as a static member of another top-level class. Unlike an inner class, a top-level class cannot directly use the instance variables of any other class. The ability to nest classes in this way allows any top-level class to provide a package-style organization for a logically related group of secondary top-level classes. Following is a simple example of an inner class:

```
public class Outer {
    int x, y;

    public int calcArea() {
        return x * y;
    }
    class Inner {
        int z;
        public int calcVolume() {
            return calcArea() * z;
        }
    }
}
```

In this example, an inner class named Inner is declared within a class called Outer. As you can see, the inner class declaration looks just like a normal (outer) class declaration. this example gives you an idea of how inner classes are structured.



## Inheritance

### Basics

What happens if you want an object that is very similar to one you already have, but that has a few extra characteristics? You just inherit a new class based on the class of the similar object. Inheritance is the process of creating a new class with the characteristics of an existing class, along with additional characteristics unique to the new class. Inheritance provides a powerful and natural mechanism for organizing and structuring programs.

So far, the discussion of classes has been limited to the data and methods that make up a class. Based on this understanding, all classes are built from scratch by defining all the data and all the associated methods.

Inheritance provides a means to create classes based on other classes.

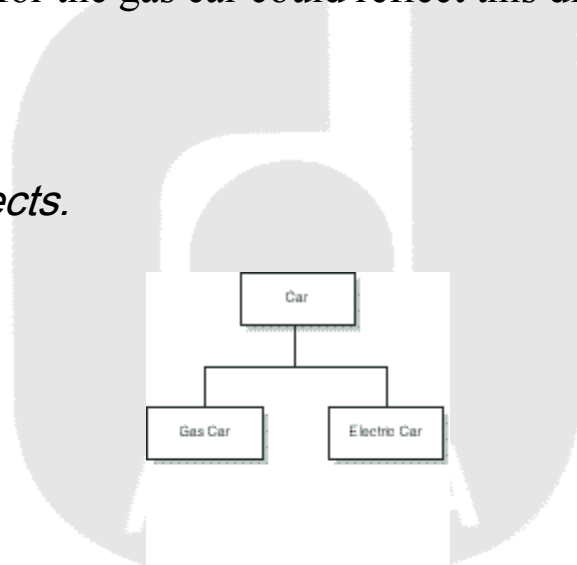
When a class is based on another class, it inherits all the properties of that class, including the data and methods for the class. The class doing the inheriting is referred to as the subclass (or the child class), and the class providing the information to inherit is referred to as the superclass (or the parent class).

Using the car example, child classes could be inherited from the car class for gas-powered cars and cars powered by electricity. Both new car classes share common "car" characteristics, but they also add a few characteristics of their own. The gas car would add, among other things,

a fuel tank and a gas cap; the electric car would add a battery and a plug for recharging. Each subclass inherits state information (in the form of variable declarations) from the superclass. Figure 5.3 shows the car parent class with the gas and electric car child classes.

Inheriting the state and behaviors of a superclass alone wouldn't do all that much for a subclass. The real power of inheritance is the ability to inherit properties and methods and add new ones; subclasses can add variables and methods to the ones they inherited from the superclass. Remember that the electric car added a battery and a recharging plug. Additionally, subclasses have the ability to override inherited methods and provide different implementations for them. For example, the gas car would probably be able to go much faster than the electric car. The accelerate method for the gas car could reflect this difference.

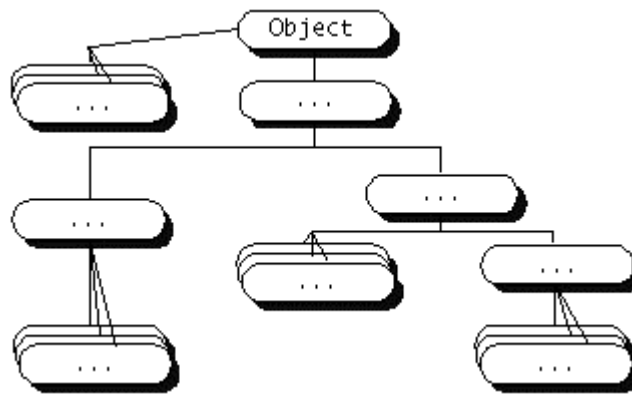
*Inherited car objects.*



**extends keyword declares that your class is a subclass of another**

You can specify only one superclass for your class (Java does not support multiple class inheritance).. So, every class in Java has one and only one immediate superclass

As depicted in the following figure, the top-most class, the class from which all other classes are derived, is the Object class defined in java.lang.



The **Object** class defines and implements behavior that every class in the Java system needs. It is the most general of all classes. Its immediate subclasses, and other classes near top of the hierarchy, implement general behavior; classes near the bottom of the hierarchy provide for more specialized behavior

The following list itemizes the members that are inherited by a subclass:

- Subclasses inherit those superclass members declared as **public** or **protected**.
- Subclasses inherit those superclass members declared with no access specifier as long as the subclass is in the same package as the superclass.
- Subclasses don't inherit a superclass's member if the subclass declares a member with the same name. In the case of member variables, the member variable in the subclass hides the one in the superclass. In the case of methods, the method in the subclass overrides the one in the superclass.

Creating a subclass can be as simple as including the **extends** clause in your class declaration. However, you usually have to make other provisions in your code when subclassing a class, such as overriding methods or providing implementations for abstract methods.

```
class W {
    float f;
}

class X extends W {
    StringBuffer sb;
}

class Y extends X {
    String s;
}

class Z extends Y {
    Integer i;
}

class Wxyz {
    public static void main(String args[]) {
        Z z = new Z();
        z.f = 4.567f;
        z.sb = new StringBuffer("abcde");
        z.s = "Teach Yourself Java";
        z.i = new Integer(41);
        System.out.println("z.f = " + z.f);
        System.out.println("z.sb = " + z.sb);
        System.out.println("z.s = " + z.s);
        System.out.println("z.i = " + z.i);
    }
}
```



## **Modifiers**

Access to variables and methods in Java classes is accomplished through access modifiers. Access modifiers define varying levels of access between class members and the outside world (other objects). Access modifiers are declared immediately before the type of a member variable or the return type of a method. There are four access modifiers: the default access modifier, public, protected, and private.

Access modifiers affect not only the visibility of class members, but also of classes themselves.

### **The Default Access Modifier**

The default access modifier specifies that only classes in the same package can have access to a class's variables and methods. Class members with default access have a visibility limited to other classes within the same package. There is no actual keyword for declaring the default access modifier; it is applied by default in the absence of an access modifier. For example, the Alien class members all had default access because no access modifiers were specified. Examples of a default access member variable and method follow:

```
long length;  
void getLength() {  
    return length;  
}
```

Notice that neither the member variable nor the method supply an access modifier, so they take on the default access modifier implicitly.

### **The public Access Modifier**

The public access modifier specifies that class variables and methods are accessible to anyone, both inside and outside the class. This means that public class members have global visibility and can be accessed by any other objects. Some examples of public member variables follow:

```
public int count;  
public boolean isActive;
```

### **The protected Access Modifier**

The protected access modifier specifies that class members are accessible only to methods in that class and subclasses of that class. This means that protected class members have visibility limited to subclasses. Examples of a protected variable and a protected method follow:

```
protected char middleInitial;  
protected char getMiddleInitial() {  
    return middleInitial;  
}
```

### **The private Access Modifier**

The private access modifier is the most restrictive; it specifies that class members are accessible only by the class in which they are defined. This means that no other class has access to private class members, even subclasses. Some examples of private member variables follow:

```
private String firstName;  
private double howBigIsIt;
```

### **The static Modifier**

There are times when you need a common variable or method for all objects of a particular class. The static modifier specifies that a variable or method is the same for all objects of a particular class.

Typically, new variables are allocated for each instance of a class. When a variable is declared as being static, it is allocated only once regardless of how many objects are instantiated. The result is that all instantiated objects share the same instance of the static variable. Similarly, a static method is one whose implementation is exactly the same for all objects of a particular class. This means that static methods have access only to static variables.

Following are some examples of a static member variable and a static method:

```
static int refCount;
```

```
static int getRefCount() {  
    return refCount;  
}
```

A beneficial side effect of static members is that they can be accessed without having to create an instance of a class. Remember the `System.out.println()` method used in the last chapter? Do you recall ever instantiating a `System` object? Of course not. `out` is a static member variable of the `System` class, which means that you can access it without having to actually instantiate a `System` object.

### **The final Modifier**

Another useful modifier in regard to controlling class member usage is the `final` modifier. The `final` modifier specifies that a variable has a constant value or that a method cannot be overridden in a subclass. To think of the `final` modifier literally, it means that a class member is the final version allowed for the class.

Following are some examples of final member variables:

```
final public int numDollars = 25;
```

```
final boolean amIBroke = false;
```

If you are coming from the world of C++, `final` variables may sound familiar. In fact, `final` variables in Java are very similar to `const` variables in C++; they must always be initialized at declaration and their value can't change any time afterward

### **The synchronized Modifier**

The `synchronized` modifier is used to specify that a method is thread safe. This means that only one path of execution is allowed into a `synchronized` method at a time. In a multithreaded environment like Java, it is possible to have many different paths of execution running through the same code. The `synchronized` modifier changes this rule by allowing only a single thread access to a method at once, forcing the other threads to wait their turn.

## The native Modifier

The native modifier is used to identify methods that have native implementations. The native modifier informs the Java compiler that a method's implementation is in an external C file. It is for this reason that native method declarations look different from other Java methods; they have no body. Following is an example of a native method declaration:

```
native int calcTotal();
```

Notice that the method declaration simply ends in a semicolon; there are no curly braces containing Java code.

## The *super* keyword

If your method hides one of its superclass's member variables, your method can refer to the hidden variable through the use of the **super** keyword. Similarly, if your method overrides one of its superclass's methods, your method can invoke the overridden method through the use of the **super** keyword.

```
class ASillyClass {
    boolean aVariable;
    void aMethod() {
        aVariable = true;
    }
}
```

and its subclass which hides aVariable and overrides aMethod:

```
class ASillierClass extends ASillyClass {
    boolean aVariable;
    void aMethod() {
        aVariable = false;
        super.aMethod();
        System.out.println(aVariable);
        System.out.println(super.aVariable);
    }
}
```

First aMethod sets aVariable (the one declared in ASillierClass that hides the one declared in ASillyClass) to false. Next aMethod invoked its overridden method with this statement:

```
super.aMethod();
```

This sets the hidden version of the aVariable (the one declared in ASillyClass) to true. Then aMethod displays both versions of aVariable which have different values:

false

true

```
class M100 {  
    int i = 100;  
}
```

```
class M200 extends M100 {  
    int i = 200;  
    void display() {  
        System.out.println("i = " + i);  
        System.out.println("super.i = " + super.i);  
    }  
}
```

```
class SuperKeyword {  
    public static void main(String args[]) {  
        M200 m200 = new M200();  
        m200.display();  
    }  
}
```

### **Constructor's hierarchy**

```
class S1 {  
    int s1;  
    S1() {
```

```
        System.out.println("S1 Constructor");
        s1 = 1;
    }
}

class T1 extends S1 {
    int t1;
    T1() {
        System.out.println("T1 Constructor");
        t1 = 2;
    }
}

class U1 extends T1 {
    int u1;
    U1() {
        System.out.println("U1 Constructor");
        u1 = 3;
    }
}

class InheritanceAndConstructors1 {
    public static void main(String args[]) {
        U1 u1 = new U1();
        System.out.println("u1.s1 = " + u1.s1);
        System.out.println("u1.t1 = " + u1.t1);
        System.out.println("u1.u1 = " + u1.u1);
    }
}
```

OutPut ::

S1 Constructor  
t1 Constructor  
u1 Constructor

```
u.s1=1  
u.t1=2  
u.u1=3
```

## Overriding

### Methods

The ability of a subclass to override a method in its superclass allows a class to inherit from a superclass whose behavior is "close enough" and then override methods as needed.

For example, all classes are descendents of the Object class. Object contains the toString method, which returns a String object containing the name of the object's class and its hash code. Most, if not all, classes will want to override this method and print out something meaningful for that class.

Let's resurrect the Stack class example and override the toString method. The output of toString should be a textual representation of the object. For the Stack class, a list of the items in the stack would be appropriate

```
public class Stack  
{  
    private Vector items;  
  
    // code for Stack's methods and constructor not shown  
  
    // overrides Object's toString method  
    public String toString() {  
        int n = items.size();  
        StringBuffer result = new StringBuffer();  
        result.append("[");  
        for (int i = 0; i < n; i++) {  
            result.append(items.elementAt(i).toString());  
            if (i < n-1) result.append(",");  
        }  
    }  
}
```

```

    }
    result.append("]");
    return result.toString();
}
}

```

The return type, method name, and number and type of the parameters for the overriding method must match those in the overridden method. The overriding method can have a different throws clause as long as it doesn't declare any types not declared by the throws clause in the overridden method. Also, the access specifier for the overriding method can allow more access than the overridden method, but not less. For example, a protected method in the superclass can be made public but not private.

### Calling the Overridden Method

Sometimes, you don't want to completely override a method. Rather, you want to add more functionality to it. To do this, simply call the overridden method using the `super` keyword. For example,

```
super.overriddenMethodName();
```

### Variables

Consider the following superclass and subclass pair:

```

class Super {
    Float aNumber;
}
class Subbie extends Super {
    Float aNumber;
}

```

The `aNumber` variable in `Subbie` hides `aNumber` in `Super`. But you can access `Super`'s `aNumber` from `Subbie` with



super.aNumber

super is a Java language keyword that allows a method to refer to hidden variables and overridden methods of the superclass.

## Abstract Classes

Sometimes, a class that you define represents an abstract concept and, as such, should not be instantiated. Take, for example, food in the real world. Have you ever seen an instance of food? No. What you see instead are instances of carrot, apple, and (our favorite) chocolate. Food represents the abstract concept of things that we all can eat. It doesn't make sense for an instance of food to exist.

Similarly in object-oriented programming, you may want to model an abstract concept without being able to create an instance of it. For example, the Number class in the java.lang package represents the abstract concept of numbers. It makes sense to model numbers in a program, but it doesn't make sense to create a generic number object. Instead, the Number class makes sense only as a superclass to classes like Integer and Float, both of which implement specific kinds of numbers. A class such as Number, which represents an abstract concept and should not be instantiated, is called an *abstract class*. An abstract class is a class that can only be subclassed-- it cannot be instantiated. To declare that your class is an abstract class, use the keyword abstract before the class keyword in your class declaration:

```
abstract class Number {  
    . . .  
}
```

If you attempt to instantiate an abstract class, the compiler displays an error similar to the following and refuses to compile your program:

```
AbstractTest.java:6: class AbstractTest is an abstract class.
```

It can't be instantiated.

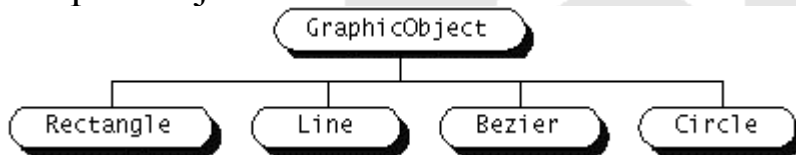
```
    new AbstractTest();  
    ^
```

1 error

## Abstract Method

An abstract class may contain *abstract methods*, that is, methods with no implementation. In this way, an abstract class can define a complete programming interface, thereby providing its subclasses with the method declarations for all of the methods necessary to implement that programming interface. However, the abstract class can leave some or all of the implementation details of those methods up to its subclasses.

Let's look at an example of when you might want to create an abstract class with an abstract method in it. In an object-oriented drawing application, you can draw circles, rectangles, lines, Bezier curves, and so on. Each of these graphic objects share certain states (position, bounding box) and behavior (move, resize, draw). You can take advantage of these similarities and declare them all to inherit from the same parent object--GraphicObject.



However, the graphic objects are also substantially different in many ways: drawing a circle is quite different from drawing a rectangle. The graphics objects cannot share these types of states or behavior. On the other hand, all GraphicObjects must know how to draw themselves; they just differ in how they are drawn. This is a perfect situation for an abstract superclass.

First you would declare an abstract class, GraphicObject, to provide member variables and methods that were wholly shared by all subclasses, such as the current position and the moveTo method. GraphicObject also declares abstract methods for methods, such as draw, that need to be implemented by all subclasses, but are implemented in entirely different ways (no default implementation in the superclass makes sense). The GraphicObject class would look something like this:

```

abstract class GraphicObject {

```

```

int x, y;
...
void moveTo(int newX, int newY) {
    ...
}
abstract void draw();
}

```

Each non-abstract subclass of `GraphicObject`, such as `Circle` and `Rectangle`, would have to provide an implementation for the `draw` method.

```

class Circle extends GraphicObject {
    void draw() {
        ...
    }
}
class Rectangle extends GraphicObject {
    void draw() {
        ...
    }
}

```

An abstract class is not required to have an abstract method in it. But any class that has an abstract method in it or that does not provide an implementation for any abstract methods declared in its superclasses *must* be declared as an abstract class.

### Using *final* and *Static*

A subclass cannot override methods that are declared `final` in the superclass (by definition, `final` methods cannot be overridden). If you attempt to override a `final` method, the compiler displays an error message similar to the following and refuses to compile the program:

FinalTest.java:7: Final methods can't be overridden.

Method void iamfinal() is final in class ClassWithFinalMethod.

```
void iamfinal() {
```

```
    ^
```

```
1 error
```

Also, a subclass cannot override methods that are declared **static** in the superclass. In other words, a subclass cannot override a class method. A subclass can *hide* a **static** method in the superclass by declaring a **static** method in the subclass with the same signature as the **static** method in the superclass.

## Packages & Interfaces

### Packages

#### Defining a package

**Definition:** A package is a collection of related classes and interfaces providing access protection and namespace management.

The syntax for the package statement follows:

```
package Identifier;
```

The classes and interfaces that are part of the Java platform are members of various packages that bundle classes by function: fundamental classes are in `java.lang`, classes for reading and writing (input and output) are in `java.io`, and so on. You can put your classes and interfaces in packages, too.

Let's look at a set of classes and examine why you might want to put them in a package. Suppose that you write a group of classes that represent a collection of graphic objects, such as circles, rectangles, lines, and points. You also write an interface, `Draggable`, that classes implement if they can be dragged with the mouse by the user:

*//in the Graphic.java file*

```
public abstract class Graphic {
    ...
}
```

*//in the Circle.java file*

```
public class Circle extends Graphic implements Draggable {
    ...
}
```

*//in the Rectangle.java file*

```
public class Rectangle extends Graphic implements Draggable {
    ...
}
```

*//in the Draggable.java file*

```
public interface Draggable {
    ...
}
```

You should bundle these classes and the interface in a package for several reasons:

You and other programmers can easily determine that these classes and interfaces are related.

You and other programmers know where to find classes and interfaces that provide graphics-related functions.

The names of your classes won't conflict with class names in other packages, because the package creates a new namespace. You can allow classes within the package to have unrestricted access to one another yet still restrict access for classes outside the package.

## Interfaces

### Basics

An interface defines a protocol of behavior that can be implemented by any class anywhere in the class hierarchy. An interface defines a set of methods but does not implement them. A class that implements the interface agrees to implement all the methods defined in the interface, thereby agreeing to certain behavior.

**Definition:** An interface is a named collection of method definitions (without implementations). An interface can also declare constants. Because an interface is simply a list of unimplemented, and therefore abstract, methods, you might wonder how an interface differs from an abstract class. The differences are significant.

- An interface cannot implement any methods, whereas an abstract class can.
- A class can implement many interfaces but can have only one superclass.
- An interface is not part of the class hierarchy. Unrelated classes can implement the same interface.

### Interface References

When you define a new interface, you are defining a new reference data type. You can use interface names anywhere you can use any other data type name. data type for the first argument to the `watchStock` method in the `StockMonitor` class is `StockWatcher`:

```
public class StockMonitor {
```

```
public void watchStock(StockWatcher  
watcher,  
String tickerSymbol, double  
delta) {  
    ...  
}  
}
```

Only an instance of a class that implements the interface can be assigned to a reference variable whose type is an interface name. So only instances of a class that implements the `StockWatcher` interface can register to be notified of stock value changes. `StockWatcher` objects are guaranteed to have a `valueChanged` method.

## Applying Interfaces

An interface defines a protocol of behavior. A class that implements an interface adheres to the protocol defined by that interface. To declare a class that implements an interface, include an `implements` clause in the class declaration. Your class can implement more than one

interface (the Java platform supports multiple inheritance for interfaces), so the implements keyword is followed by a comma-separated list of the interfaces implemented by the class.

**By Convention:** The implements clause follows the extends clause, if it exists.

Here's a partial example of an applet that implements the StockWatcher interface:

```
public class StockApplet extends Applet
implements StockWatcher {
    ...

    public void valueChanged(String
tickerSymbol, double newValue) {
        if (tickerSymbol.equals(sunTicker)) {
            ...
        } else if
(tickerSymbol.equals(oracleTicker)) {
            ...
        } else if
(tickerSymbol.equals(ciscoTicker)) {
```



```

        ...
    }
}
}

```

Note that this class refers to each constant defined in `StockWatcher`, `sunTicker`, `oracle-Ticker`, `imple name`.

Classes that implement an interface inherit the constants defined within that interface. So those classes can use simple names to refer to the constants. Any other class can use an interfaces constants with a qualified name, like this:

`StockWatcher.sunTicker`

When a class implements an interface, it is essentially signing a contract. Either the class must implement all the methods declared in the interface and its superinterfaces, or the class must be declared abstract. The method signature--the name and the number and type of arguments in the class--must match the method signature as it appears in the interface. The `StockApplet` implements the `StockWatcher` interface, so the applet provides an implementation for the `valueChanged` method. The method ostensibly updates the applets display or otherwise uses this information.

## **Interface variables**

The interface body contains method declarations for all the methods included in the interface. A method declaration within an interface is followed by a semicolon (;) because an interface does not provide implementations for the methods declared within it. All methods declared in an interface are implicitly public and abstract.

An interface can contain constant declarations in addition to method declarations. All constant values defined in an interface are implicitly public, static, and final.

Member declarations in an interface disallow the use of some declaration modifiers; you cannot use transient, volatile, or synchronized in a member declaration in an interface. Also, you may not use the private and protected specifiers when declaring members of an interface.

## **Interface Inheritance**

An interface can extend other interfaces, just as a class can extend or subclass another class. However, whereas a class can extend only one other class, an interface can extend any number of interfaces. The list of superinterfaces is a comma-separated list of all the interfaces extended by the new interface.

## Exception Handling

### Basics

The term *exception* is shorthand for the phrase "exceptional event." It can be defined as follows:

**Definition:** An *exception* is an event that occurs during the execution of a program that disrupts the normal flow of instructions.

Many kinds of errors can cause exceptions--problems ranging from serious hardware errors, such as a hard disk crash, to simple programming errors, such as trying to access an out-of-bounds array element. When such an error occurs within a Java method, the method creates an exception object and hands it off to the runtime system. The exception object contains information about the exception, including its type and the state of the program when the error occurred. The runtime system is then responsible for finding some code to handle the error. In Java terminology, creating an exception object and handing it to the runtime system is called *throwing an exception*

Often exceptions fall into categories or groups. For example, you could imagine a group of exceptions, each of which represents a specific type of error that can occur when manipulating an array: the index is out of range for the size of the array, the element being inserted into the array is of the wrong type, or the element being searched for is not in the array. Furthermore, you can imagine that some methods would like to handle all exceptions that fall within a category (all array exceptions), and other methods would like to handle specific exceptions (just the invalid index exceptions, please).

Because all exceptions that are thrown within a Java program are first-class objects, grouping or categorization of exceptions is a natural outcome of the class hierarchy. Java exceptions must be

instances of Throwable or any Throwable descendant. As for other Java classes, you can create subclasses of the Throwable class and subclasses of your subclasses.

## Exception & Error Classes

If you have done any amount of Java programming at all, you have undoubtedly already encountered exceptions. Your first encounter with Java exceptions was probably in the form of an error message from the compiler like this one:

```
InputFile.java:11: Exception
java.io.FileNotFoundException
must be caught, or it must be declared
in the throws clause
of this method.
```

```
    in = new FileReader(filename);
        ^
```

This message indicates that the compiler found an exception that is not being handled. The Java language requires that a method either catch all "checked" exceptions (those that are checked by the runtime system) or specify that it can throw that type of exception

## Class Throwable

### Errors

When a dynamic linking failure or some other "hard" failure in the virtual machine occurs, the virtual machine throws an Error.

Typical Java programs should not catch Errors. In addition, it's unlikely that typical Java programs will ever throw Errors either.

## Exceptions

Most programs throw and catch objects that derive from the Exception class. Exceptions indicate that a problem occurred but that the problem is not a serious systemic problem. Most programs you write will throw and catch Exceptions.

The Exception class has many descendants defined in the Java packages. These descendants indicate various types of exceptions that can occur. For example, `IllegalAccess` signals that a particular method could not be found, and `NegativeArraySizeException` indicates that a program attempted to create an array with a negative size.

### Checked Exceptions

Java has different types of exceptions, including I/O Exceptions, runtime exceptions, and exceptions of your own creation, to name a few. Of interest to us in this discussion are runtime exceptions. Runtime exceptions are those exceptions that occur within the Java runtime system. This includes arithmetic exceptions (such as when dividing by zero), pointer exceptions (such as trying to access an object through a null reference), and indexing exceptions (such as attempting to access an array element through an index that is too large or too small).

Runtime exceptions can occur anywhere in a program and in a typical program can be very numerous. The cost of checking for runtime exceptions often exceeds the benefit of catching or specifying them. Thus the compiler does not require that you catch or specify runtime exceptions, although you can. *Checked exceptions* are exceptions that are not runtime exceptions and are checked by the compiler; the compiler checks that these exceptions are caught or specified.

## Using try ... catch

The following example defines and implements a class named `ListOfNumbers`. The `ListOfNumbers` class calls two methods from classes in the Java packages that can throw exceptions.

```
// Note: This class won't compile by design!  
// See ListOfNumbersDeclared.java or  
// ListOfNumbers.java  
// for a version of this class that will compile.  
import java.io.*;  
import java.util.Vector;  
  
public class ListOfNumbers {  
    private Vector vector;  
    private static final int size = 10;  
  
    public ListOfNumbers () {  
        vector = new Vector(size);  
        for (int i = 0; i < size; i++)  
            vector.addElement(new Integer(i));  
    }  
    public void writeList() {  
        PrintWriter out = new PrintWriter(new  
        FileWriter("OutFile.txt"));  
  
        for (int i = 0; i < size; i++)  
            out.println("Value at: " + i + " = " +  
            vector.elementAt(i));  
  
        out.close();  
    }  
}
```

Upon construction, `ListOfNumbers` creates a `Vector` that contains ten `Integer` elements with sequential values 0 through 9. The `ListOfNumbers` class also defines a method named `writeList` that writes the list of numbers into a text file called `OutFile.txt`.

The `writeList` method calls two methods that can throw exceptions. First, the following line invokes the constructor for `FileWriter`, which throws an `IOException` if the file cannot be opened for any reason:

```
out = new PrintWriter(new  
    FileWriter("OutFile.txt"));
```

Second, the `Vector` class's `elementAt` method throws an `ArrayIndexOutOfBoundsException` if you pass in an index whose value is too small (a negative number) or too large (larger than the number of elements currently contained by the `Vector`). Here's how `ListOfNumbers` invokes `elementAt`:

```
out.println("Value at: " + i + " = " +  
    vector.elementAt(i));
```

The first step in constructing an exception handler is to enclose the statements that might throw an exception within a try block. In general, a try block looks like this:

```
try {  
    Java statements  
}
```

The segment of code labelled *Java statements* is composed of one or more legal Java statements that could throw an exception

To construct an exception handler for the `writeList` method from the `ListOfNumbers` class, you need to enclose the exception-throwing statements of the `writeList` method within a try block. There is more than one way to accomplish this task. You could put each statement that might potentially throw an exception within its own try statement, and provide separate exception handlers for each try. Or you could put all of the `writeList` statements within a single try statement and associate multiple handlers with it. The following listing uses one try statement for the entire method because the code tends to be easier to read.

```
PrintWriter out = null;

try {
    System.out.println("Entering try statement");
    out = new PrintWriter(
        new FileWriter("OutFile.txt"));

    for (int i = 0; i < size; i++)
        out.println("Value at: " + i + " = " +
            victor.elementAt(i));
}
```

The try statement governs the statements enclosed within it and defines the scope of any exception handlers associated with it. In other words, if an exception occurs within the try statement, that exception is handled by the appropriate exception handler associated with this try statement.

A try statement *must* be accompanied by at least one catch block or one finally block



## Multiple Catch Statements

The catch block contains a series of legal Java statements. These statements are executed if and when the exception handler is invoked. The runtime system invokes the exception handler when the handler is the first one in the call stack whose type matches that of the exception thrown.

The writeList method from the ListOfNumbers class uses two exception handlers for its try statement, with one handler for each of the two types of exceptions that can be thrown within the try block -- `ArrayIndexOutOfBoundsException` and `IOException`.

```
try {  
    ...  
} catch (ArrayIndexOutOfBoundsException e) {  
    System.err.println("Caught  
ArrayIndexOutOfBoundsException: " +  
        e.getMessage());  
} catch (IOException e) {  
    System.err.println("Caught IOException: " +  
        e.getMessage());  
}
```

## Throw, Throws & Finally Statements

All Java methods use the throw statement to throw an exception. The throw statement requires a single argument: a *throwable* object. In the Java system, throwable objects are instances of any subclass of the `Throwable` class. Here's an example of a throw statement:

```
throw someThrowableObject;
```

If you attempt to throw an object that is not throwable, the compiler refuses to compile your program and displays an error message similar to the following:

```
testing.java:10: Cannot throw class java.lang.Integer;
it must be a subclass of class java.lang.Throwable.
```

```
    throw new Integer(4);
```

```
    ^
```

The following method is taken from a class that implements a common stack object. The pop method removes the top element from the stack and returns it:

```
public Object pop() throws EmptyStackException {
    Object obj;

    if (size == 0)
        throw new EmptyStackException();

    obj = objectAt(size - 1);
    setObjectAt(size - 1, null);
    size--;
    return obj;
}
```

The pop method checks to see if there are any elements on the stack. If the stack is empty (its size is equal to 0), then pop instantiates a new EmptyStackException object and throws it. The EmptyStackException class is defined in the java.util package.

Later pages really need to remember is that you can throw only objects that inherit from the `java.lang.Throwable` class.

You'll notice that the declaration of the `pop` method contains this clause:

```
throws EmptyStackException
```

The `throws` clause specifies that the method can throw an `EmptyStackException`. As you know, the Java language requires that methods either catch or specify all checked exceptions that can be thrown within the scope of that method. You do this with the `throws` clause of the method declaration

## The finally Block

Java's `finally` block provides a mechanism that allows your method to clean up after itself regardless of what happens within the `try` block. Use the `finally` block to close files or release other system resources.

The runtime system always executes the statements within the `finally` block regardless of what happens within the `try` block. Regardless of whether control exits the method's `try` block, the code within the `finally` block will be executed.

This is the `finally` block for the `writeList` method. It cleans up and closes the `PrintWriter`.

```
finally {  
    if (out != null) {
```

```
System.out.println("Closing PrintWriter");

        out.close();
    } else {
        System.out.println("PrintWriter not open");
    }
}
```

### **Java's Built in Exception**

AcINotFoundException, ActivationException,  
AlreadyBoundException, ApplicationException, AWTException,  
BadLocationException, ClassNotFoundException,  
CloneNotSupportedException, DataFormatException,  
ExpandVetoException, FontFormatException,  
GeneralSecurityException, IllegalAccessException,  
InstantiationException, InterruptedException,  
IntrospectionException, InvalidMidiDataException,  
InvocationTargetException, IOException, LastOwnerException,  
LineUnavailableException, MidiUnavailableException,  
MimeTypeParseException, NamingException,  
NoninvertibleTransformException, NoSuchFieldException,  
NoSuchMethodException, NotBoundException,  
NotOwnerException, ParseException, PrinterException,  
PrivilegedActionException, PropertyVetoException,  
RemarshalException, RuntimeException,  
ServerNotActiveException, SQLException,  
TooManyListenersException, UnsupportedAudioFileException,  
UnsupportedFlavorException,  
UnsupportedLookAndFeelException, UserException

## **Java.lang.\***

### **Interfaces**

#### **Cloneable**

A class implements the Cloneable interface to indicate to the Object.clone() method that it is legal for that method to make a field-for-field copy of instances of that class.

Attempts to clone instances that do not implement the Cloneable interface result in the exception CloneNotSupportedException being thrown.

The interface Cloneable declares no methods.

#### **Comparable**

This interface imposes a total ordering on the objects of each class that implements it. This ordering is referred to as the class's natural ordering, and the class's compareTo method is referred to as its natural comparison method.

Lists (and arrays) of objects that implement this interface can be sorted automatically by Collections.sort (and Arrays.sort). Objects that implement this interface can be used as keys in a sorted map or elements in a sorted set, without the need to specify a comparator.

A class's natural ordering is said to be consistent with equals if and only if `(e1.compareTo((Object)e2)==0)` has the same boolean value as `e1.equals((Object)e2)` for every `e1` and `e2` of class `C`.

It is strongly recommended (though not required) that natural orderings be consistent with equals. This is so because sorted sets (and sorted maps) without explicit comparators behave "strangely" when they are used with elements (or keys) whose natural ordering is inconsistent with equals. In particular, such a sorted set (or sorted map) violates the general contract for set (or map), which is defined in terms of the equals operation.

For example, if one adds two keys *a* and *b* such that `(a.equals((Object)b) && a.compareTo((Object)b) != 0)` to a sorted set that does not use an explicit comparator, the second add operation returns false (and the size of the sorted set does not increase) because *a* and *b* are equivalent from the sorted set's perspective.

Virtually all Java core classes that implement comparable have natural orderings that are consistent with equals. One exception is `java.math.BigDecimal`, whose natural ordering equates BigDecimals with equal values and different precisions (such as 4.0 and 4.00).

For the mathematically inclined, the relation that defines the natural ordering on a given class *C* is:

$$\{(x, y) \text{ such that } x.compareTo((Object)y) \leq 0\}.$$

The quotient for this total order is:

$$\{(x, y) \text{ such that } x.compareTo((Object)y) == 0\}.$$

It follows immediately from the contract for `compareTo` that the quotient is an equivalence relation on *C*, and that the natural ordering is a total order on *C*. When we say that a class's natural ordering is consistent with equals, we mean that the quotient for the natural ordering is the equivalence relation defined by the class's `equals(Object)` method:

$$\{(x, y) \text{ such that } x.equals((Object)y)\}.$$

## Runnable

The `Runnable` interface provides a common approach to identifying the code to be executed as part of an active thread. It consists of a single method, `run()`, which is executed when a thread is activated. The `Runnable` interface is implemented by the `Thread` class and by other classes that support threaded execution.

## Classes

### Object

Object and Class are two of the most important classes in the Java API. The Object class is at the top of the Java class hierarchy. All classes are subclasses of Object and therefore inherit its methods. The Class class is used to provide class descriptors for all objects created during Java program execution.

The Object class does not have any variables and has only one constructor. However, it provides 11 methods that are inherited by all Java classes and that support general operations that are used with all objects. For example, the equals() and hashCode() methods are used to construct hash tables of Java objects. Hash tables are like arrays, but they are indexed by key values and dynamically grow in size. They make use of hash functions to quickly access the data that they contain. The hashCode() method creates a hash code for an object

The clone() method creates an identical copy of an object. The object must implement the Cloneable interface. This interface is defined within the java.lang package. It contains no methods and is used only to differentiate cloneable from noncloneable classes.

The getClass() method identifies the class of an object by returning an object of Class

The toString() method creates a String representation of the value of an object. This method is handy for quickly displaying the contents of an object. When an object is displayed, using print() or println(), the toString() method of its class is automatically called to convert the object into a string before printing. Classes that override the toString() method can easily provide a custom display for their objects.

The finalize() method of an object is executed when an object is garbage-collected. The method performs no action, by default, and needs

to be overridden by any class that requires specialized finalization processing.

The Object class provides three `wait()` and two `notify()` methods that support thread control. These methods are implemented by the Object class so that they can be made available to threads that are not created from subclasses of class Thread. The `wait()` methods cause a thread to wait until it is notified or until a specified amount of time has elapsed. The `notify()` methods are used to notify waiting threads that their wait is over.

## **Number**

The Number class is an abstract numeric class that is subclassed by Integer, Long, Float, and Double. It provides four methods that support conversion of objects from one class to another.

## **Wrapper Classes**

Variables that are declared using the primitive Java types are not objects and cannot be created and accessed using methods. Primitive types also cannot be subclassed. To get around the limitations of primitive types, the `java.lang` package defines class *wrappers* for these types.

## **Byte and Short**

The Byte and Short classes wrap the Byte and Short primitive types. They provide the `MIN_VALUE` and `MAX_VALUE` constants, as well as a number of type and class testing and conversion methods. The `parseByte()` and `parseShort()` methods are used to parse String objects and convert them to Integer and Long objects.



## **Integer and Long**

The Integer and Long classes wrap the int and long primitive types. They provide the `MIN_VALUE` and `MAX_VALUE` constants, as well as a number of type and class testing and conversion methods. The `parseInt()` and `parseLong()` methods are used to parse String objects and convert them to Integer and Long objects.

## **Float and Double**

The Double and Float classes wrap the double and float primitive types. They provide the `MIN_VALUE`, `MAX_VALUE`, `POSITIVE_INFINITY`, and `NEGATIVE_INFINITY` constants, as well as the NaN (not-a-number) constant. NaN is used as a value that is not equal to any value, including itself. These classes provide a number of type and class testing and conversion methods, including methods that support conversion to and from integer bit representations.

## **Character**

The Character class is a wrapper for the char primitive type. It provides several methods that support case, type, and class testing, and conversion. Check out the API pages on these methods. We'll use some of them in the upcoming example.

## **Boolean**

The Boolean class is a wrapper for the boolean primitive type. It provides the `getBoolean()`, `toString()`, and `booleanValue()` methods to support type and class conversion. The `toString()`, `equals()`, and `hashCode()` methods override those of class Object.

## **Class**

The `Class` class provides eight methods that support the runtime processing of an object's class and interface information. This class does not have a constructor. Objects of this class, referred to as class descriptors, are automatically created and associated with the objects to which they refer. Despite their name, class descriptors are used for interfaces as well as classes.

The `getName()` and `toString()` methods return the `String` containing the name of a class or interface. The `toString()` method differs in that it prepends the string `class` or `interface`, depending on whether the class descriptor is a class or an interface. The static `forName()` method is used to obtain a class descriptor for the class specified by a `String` object.

The `getSuperclass()` method returns the class descriptor of the superclass of a class. The `isInterface()` method identifies whether a class descriptor applies to a class or an interface. The `getInterface()` method returns an array of `Class` objects that specify the interfaces of a class, if any.

The `newInstance()` method creates an object that is a new instance of the specified class. It can be used in lieu of a class's constructor, although it is generally safer and clearer to use a constructor rather than `newInstance()`.

The `getClassLoader()` method returns the class loader of a class, if one exists. Classes are not usually loaded by a class loader. However, when a class is loaded from outside the `CLASSPATH`, such as over a network, a class loader is used to convert the class byte stream into a class descriptor. The `ClassLoader` class is covered later in this .

## **Math**

The `Math` class provides an extensive set of mathematical methods in the form of a static class library. It also defines the mathematical constants `E` and `PI`. The supported methods include arithmetic, trigonometric, exponential, logarithmic, random number, and conversion routines. You should browse the API page of this class to get a feel for the methods it provides. The example only touches on a few of these methods.

The source code of the MathApp program.

```
import java.lang.System;
import java.lang.Math;

public class MathApp {
    public static void main(String args[]) {
        System.out.println(Math.E);
        System.out.println(Math.PI);
        System.out.println(Math.abs(-1234));
        System.out.println(Math.cos(Math.PI/4));
        System.out.println(Math.sin(Math.PI/2));
        System.out.println(Math.tan(Math.PI/4));
        System.out.println(Math.log(1));
        System.out.println(Math.exp(Math.PI));
        for(int i=0;i<5;++i)
            System.out.print(Math.random()+" ");
        System.out.println();
    }
}
```

This program prints the constants  $e$  and  $\pi$ ,  $|-1234|$ ,  $\cos(\pi/4)$ ,  $\sin(\pi/2)$ ,  $\tan(\pi/4)$ ,  $\ln(1)$ ,  $e^\pi$ , and then five random double numbers between 0.0 and 1.1. Its output is as follows:

```
2.71828
3.14159
1234
0.707107
1
1
0
23.1407
0.831965 0.573099 0.0268818 0.378625 0.164485
```

The random numbers you generate will almost certainly differ from the ones shown here

## **String and StringBuffer**

The String and StringBuffer classes are used to support operations on strings of characters. The String class supports constant (unchanging) strings, whereas the StringBuffer class supports growable, modifiable strings. String objects are more compact than StringBuffer objects, but StringBuffer objects are more flexible.

### **String Literals**

String literals are strings that are specified using double quotes. "This is a string" and "xyz" are examples of string literals. String literals are different than the literal values used with primitive types. When the javac compiler encounters a String literal, it converts it to a String constructor. For example, the following:

```
String str = "text";
```

is equivalent to this:

```
String str = new String("text");
```

The fact that the compiler automatically supplies String constructors allows you to use String literals everywhere that you could use objects of the String class.

### **String Constructors**

The String class provides seven constructors for the creation and initialization of String objects. These constructors allow strings to be created from other strings, string literals, arrays of characters, arrays of bytes, and StringBuffer objects. Browse through the API page for the String class to become familiar with these constructors.

## **String Access Methods**

The String class provides a very powerful set of methods for working with String objects. These methods allow you to access individual characters and substrings; test and compare strings; copy, concatenate, and replace parts of strings; convert and create strings; and perform other useful string operations.

The most important String methods are the `length()` method, which returns an integer value identifying the length of a string; the `charAt()` method, which allows the individual characters of a string to be accessed; the `substring()` method, which allows substrings of a string to be accessed; and the `valueOf()` method, which allows primitive data types to be converted into strings.

In addition to these methods, the Object class provides a `toString()` method for converting other objects to String objects. This method is often overridden by subclasses to provide a more appropriate object-to-String conversion

## **String Access Methods**

The String class provides a very powerful set of methods for working with String objects. These methods allow you to access individual characters and substrings; test and compare strings; copy, concatenate, and replace parts of strings; convert and create strings; and perform other useful string operations.

The most important String methods are the `length()` method, which returns an integer value identifying the length of a string; the `charAt()` method, which allows the individual characters of a string to be accessed; the `substring()` method, which allows substrings of a string to be accessed; and the `valueOf()` method, which allows primitive data types to be converted into strings.

In addition to these methods, the Object class provides a `toString()` method for converting other objects to String objects. This method is often overridden by subclasses to provide a more appropriate object-to-String conversion.

## Character and Substring Methods

Several String methods allow you to access individual characters and substrings of a string. These include `charAt()`, `getBytes()`, `getChars()`, `indexOf()`, `lastIndexOf()`, and `substring()`. Whenever you need to perform string manipulations, be sure to check the API documentation to make sure that you don't overlook an easy-to-use, predefined String method.

## String Comparison and Test Methods

Several String methods allow you to compare strings, substrings, byte arrays, and other objects with a given string. Some of these methods are `compareTo()`, `endsWith()`, `equals()`, `equalsIgnoreCase()`, `regionMatches()`, and `startsWith()`.

## Copy, Concatenation, and Replace Methods

The following methods are useful for copying, concatenating, and manipulating strings: `concat()`, `copyValueOf()`, `replace()`, and `trim()`.

## String Conversion and Generation

There are a number of string methods that support String conversion. These are `intern()`, `toCharArray()`, `toLowerCase()`, `toString()`, `toUpperCase()`, and `valueOf()`. You explore the use of some of these methods in the following example.

The source code of the StringApp program.

```
import java.lang.System;
import java.lang.String;

public class StringApp {
    public static void main(String args[]) {
        String s = " Java Developer's Guide ";
        System.out.println(s);
        System.out.println(s.toUpperCase());
        System.out.println(s.toLowerCase());
        System.out.println "["+s+""];
    }
}
```

```
s=s.trim();
System.out.println("[ "+s+" ]");
s=s.replace('J','X');
s=s.replace('D','Y');
s=s.replace('G','Z');
System.out.println(s);
int i1 = s.indexOf('X');
int i2 = s.indexOf('Y');
int i3 = s.indexOf('Z');
char ch[] = s.toCharArray();
ch[i1]='J';
ch[i2]='D';
ch[i3]='G';
s = new String(ch);
System.out.println(s);
}
}
```

The program's output is as follows:

```
Java Developer's Guide
JAVA DEVELOPER'S GUIDE
java developer's guide
[ Java Developer's Guide ]
[Java Developer's Guide]
Xava Yeveloper's Zuide
Java Developer's Guide
```

The StringBuffer class is the force behind the scene for most complex string manipulations. The compiler automatically declares and manipulates objects of this class to implement common string operations.

The StringBuffer class provides three constructors: an empty constructor, an empty constructor with a specified initial buffer length, and a constructor that creates a StringBuffer object from a String object.

In general, you will find yourself constructing `StringBuffer` objects from `String` objects, and the last constructor will be the one you use most often.

The `StringBuffer` class provides several versions of the `append()` method to convert and append other objects and primitive data types to `StringBuffer` objects. It provides a similar set of `insert()` methods for inserting objects and primitive data types into `StringBuffer` objects. It also provides methods to access the character-buffering capacity of `StringBuffer` and methods for accessing the characters contained in a string. It is well worth a visit to the `StringBuffer` API pages to take a look at the methods that it has to offer.

The source code of the `StringBufferApp` program.

```
import java.lang.System;
import java.lang.String;
import java.lang.StringBuffer;

public class StringBufferApp {
    public static void main(String args[]) {
        StringBuffer sb = new StringBuffer(" is ");
        sb.append("Hot");
        sb.append('!');
        sb.insert(0,"Java");
        sb.append('\n');
        sb.append("This is ");
        sb.append(true);
        sb.setCharAt(21,'T');
        sb.append('\n');
        sb.append("Java is #");
        sb.append(1);
        String s = sb.toString();
        System.out.println(s);
    }
}
```



```
}  
}
```

The program creates a StringBuffer object using the string " is ". It appends the string "Hot" using the append() method and the character '!' using an overloaded version of the same method. The insert() method is used to insert the string "Java" at the beginning of the string buffer.

Three appends are used to tack on a newline character (\n), the string "This is ", and the boolean value true. The append() method is overloaded to support the appending of the primitive data types as well as arbitrary Java objects.

The setCharAt() method is used to replace the letter 't' at index 21 with the letter 'T'. The charAt() and setCharAt() methods allow StringBuffer objects to be treated as arrays of characters.

Finally, another newline character is appended to sb, followed by the string "Java is #" and the int value 1. The StringBuffer object is then converted to a string and displayed to the console window.

The output of the program is as follows:

```
Java is Hot!  
This is True  
Java is #1
```

## System

The System class provides an interface to a number of useful system resources. Among these are the System.in and System.out input and output streams. The System.out stream was used in the preceding example. The following example illustrates the use of System.in.

This program builds on what you learned from HelloWorldApp. HelloWorldApp just displayed a message to your console window. The ICanReadApp program will read your name from the keyboard

characters you type and display it on the console window. It introduces the concepts of identifiers, variable declarations, Java keywords, and object constructors.

Use your text editor to create a file called ICanReadApp.java with the Java program.

The source code of the I Can Read! program.

```
// ICanReadApp.java

import java.lang.System;
import java.io.DataInputStream;
import java.io.IOException;
class ICanReadApp {
    public static void main (String args[]) throws IOException {
        System.out.print("Enter your name: ");
        System.out.flush();
        String name;
        DataInputStream keyboardInput = new
DataInputStream(System.in);
        name=keyboardInput.readLine();
        System.out.println("Your name is: "+name);
    }
}
```

Save the file in your c:\java\jdg\ch04 directory. Compile it with the command line

```
javac ICanReadApp.java
```

This will produce a file named ICanReadApp.class that contains the binary compiled code for your program. Run the program with the command line

```
java ICanReadApp
```

Make sure that your CLASSPATH is correctly set so that Java can find the ICanReadApp class.

The program will prompt you to enter your name. When you enter your name, the program will display it to you. Here is a sample program run:

```
C:\java\jdg\ch04>java ICanReadApp
Enter your name: Jamie
Your name is: Jamie
```

It may seem that you're going nowhere fast, but this little program illustrates some more basic Java syntax. Hang in there-by the time you get to the end of the chapter, you'll be having fun with Java console programming.

## **Thread**

The Thread class is used to construct and access individual threads of execution that are executed as part of a multithreaded program. It defines the priority constants, MIN\_PRIORITY, MAX\_PRIORITY, and NORM\_PRIORITY, that are used to control task scheduling. It provides seven constructors for creating instances of class Thread. The four constructors with the Runnable parameters are used to construct threads for classes that do not subclass the Thread class. The other constructors are used for the construction of Thread objects from Thread subclasses.

Thread supports many methods for accessing Thread objects. These methods provide the capabilities to work with a thread's group; obtain detailed information about a thread's activities; set and test a thread's properties; and cause a thread to wait, be interrupted, or be destroyed.

## **ThreadGroup**

The ThreadGroup class is used to encapsulate a group of threads as a single object so that they can be accessed as a single unit. A number of access methods are provided for manipulating ThreadGroup objects. These methods keep track of the threads and thread groups contained in a thread group and perform global operations on all threads in the group. The global operations are group versions of the operations that are provided by the Thread class.

## **Throwable**

The Throwable class is at the top of the Java error-and-exception hierarchy. It is extended by the Error and Exception classes and provides methods that are common to both classes. These methods consist of stack tracing methods, the getMessage() method, and the toString() method, which is an override of the method inherited from the Object class. The getMessage() method is used to retrieve any messages that are supplied in the creation of Throwable objects.

The fillInStackTrace() and printStackTrace() methods are used to add information to supply and print information that is used to trace the propagation of exceptions and errors throughout a program's execution.

## **The Error Class**

The Error class is used to provide a common superclass to define abnormal and fatal events that should not occur. It provides two constructors and no other methods. Four major classes of errors extend the Error class: AWTError, LinkageError, ThreadDeath, and VirtualMachineError.

The AWTError class identifies fatal errors that occur in the Abstract Window Toolkit packages. It is a single identifier for all AWT errors and is not subclassed.

The `LinkageError` class is used to define errors that occur as the result of incompatibilities between dependent classes. These incompatibilities result when a class X that another class Y depends on is changed before class Y can be recompiled. The `LinkageError` class is extensively subclassed to identify specific manifestations of this type of error.

The `ThreadDeath` error class is used to indicate that a thread has been stopped. Instances of this class can be caught and then rethrown to ensure that a thread is gracefully terminated, although this is not recommended. The `ThreadDeath` class is not subclassed.

The `VirtualMachineError` class is used to identify fatal errors occurring in the operation of the Java virtual machine. It has four subclasses: `InternalError`, `OutOfMemoryError`, `StackOverflowError`, and `UnknownError`.

## **The Exception Class**

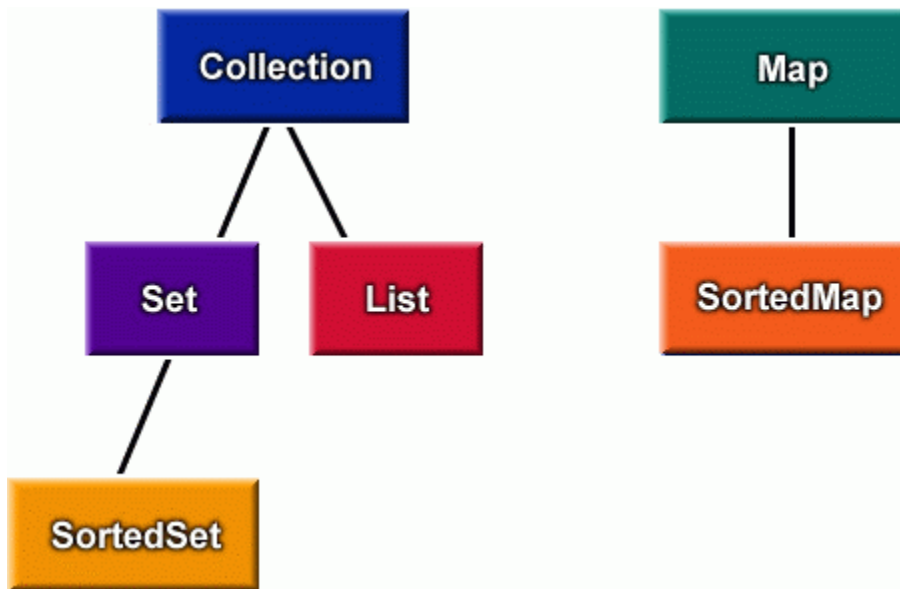
The `Exception` class provides a common superclass for the exceptions that can be defined for Java programs and applets. There are nine subclasses of exceptions that extend the `Exception` class. These exception subclasses are further extended by lower-level subclasses.

## **Java.util.\***

### **Basics**

The *core collection interfaces* are the interfaces used to manipulate collections, and to pass them from one method to another. The basic purpose of these interfaces is to allow collections to be manipulated independently of the details of their representation. The core collection interfaces are the heart and soul of the collections framework. When you understand how to use these interfaces, you know most of what there is

to know about the framework. The core collections interfaces are shown below:



## Interfaces

### Collection

The Collection interface is the root of the collection hierarchy. A **Collection** represents a group of objects, known as its *elements*. Some **Collection** implementations allow duplicate elements and others do not. Some are ordered and others unordered. The JDK doesn't provide any direct implementations of this interface: It provides implementations of more specific subinterfaces like **Set** and **List**. This interface is the least common denominator that all collections implement. **Collection** is used to pass collections around and manipulate them when maximum generality is desired.

## List

A List is an ordered collection (sometimes called a *sequence*). Lists can contain duplicate elements. The user of a **List** generally has precise control over where in the **List** each element is inserted. The user can access elements by their integer index (position). If you've used Vector , you're already familiar with the general flavor of **List**.

## Set

A Set is a collection that cannot contain duplicate elements. As you might expect, this interface models the mathematical *set* abstraction. It is used to represent sets like the cards comprising a poker hand, the courses making up a student's schedule, or the processes running on a machine.

## SortedSet

A **SortedSet** is a **Set** that maintains its elements in ascending order. Several additional operations are provided to take advantage of the ordering. The **SortedSet** interface is used for things like word lists and membership rolls.

## Map

A Map is an object that maps keys to values. Maps cannot contain duplicate keys: Each key can map to at most one value. If you've used Hashtable , you're already familiar with the general flavor of **Map**.

## SortedMap

A `SortedMap` is a `Map` that maintains its mappings in ascending key order. It is the `Map` analogue of `SortedSet`. The `SortedMap` interface is used for apps like dictionaries and telephone directories.

## Enumeration and Iterator

The object returned by the `iterator` method deserves special mention. It is an `Iterator`, which is very similar to an `Enumeration`, but differs in two respects:

- `Iterator` allows the caller to remove elements from the underlying collection during the iteration with well-defined semantics.
- Method names have been improved.

The first point is important: There was *no* safe way to remove elements from a collection while traversing it with an `Enumeration`. The semantics of this operation were ill-defined, and differed from implementation to implementation

## Classes

### All Implementing Classes of Collection & Map

## Vector

Java doesn't include dynamically linked lists, queues, or other data structures of that type. Instead, the designers of Java envisioned the `Vector` class, which handles the occasions when you need to dynamically store objects. Of course, there are positive and negative



consequences of this decision by the designers at Sun. On the positive side, the Vector class contributes to the simplicity of the language. The major negative point is that, at face value, the Vector class severely limits programmers from using more sophisticated programs.

In any case, the Vector class implements a dynamically allocated list of objects. It attempts to optimize storage by increasing the storage capacity of the list when needed by increments larger than just one object. Typically, with this mechanism, there is some excess capacity in the list. When this capacity is exhausted, the list is reallocated to add another block of objects at the end of the list. Setting the capacity of the Vector object to the needed size before inserting a large number of objects reduces the need for incremental reallocation. Because of this mechanism, it is important to remember that the capacity (the available elements in the Vector object) and the size (the number of elements currently stored in the Vector object) usually are not the same.

Suppose that a Vector with capacityIncrement equal to 3 has been created. As objects are added to the Vector, new space is allocated in chunks of three objects. After five elements have been added, there is still room for one more element without the need for any additional memory allocation.

After the sixth element has been added, there is no more excess capacity. When the seventh element is added, a new allocation is made to add three additional elements, giving a total capacity of nine. After the seventh element is added, there are two remaining unused elements.

The initial storage capacity and the capacity increment can both be specified in the constructor. Even though the capacity is automatically increased as needed, the ensureCapacity() method can be used to increase the capacity to a specific minimum number of elements; the trimToSize() method can be used to reduce the capacity to the minimum number of elements needed to store the current amount. New elements can be added to the Vector using the addElement() and insertElementAt() methods. The elements passed to be stored in the

Vector must be derived from type Object. Elements can be changed using the `setElementAt()` method. Removal of elements is accomplished with the `removeElement()`, `removeElementAt()`, and `removeAllElements()` methods. Elements can be accessed directly using the `elementAt()`, `firstElement()`, and `lastElement()` methods; elements can be located using the `indexOf()` and `lastIndexOf()` methods. Information about the size and the capacity of the Vector are returned by the `size()` and `capacity()` methods. The `setSize()` method can be used to directly change the size of the Vector.

For example, the `Vector1` code in Listing creates a Vector of integers by adding new elements to the end. Then, using a variety of techniques, it prints the Vector

```
import java.lang.Integer;
import java.util.Enumeration;
import java.util.Vector;
class Vector1 {
    public static void main(String args[]){
        Vector v=new Vector(10,10);
        for (int i=0;i<20;i++)
            v.addElement(new Integer(i));
        System.out.println("Vector in original order using an
Enumeration");
        for (Enumeration e=v.elements();e.hasMoreElements();)
            System.out.print(e.nextElement()+" ");
        System.out.println();
        System.out.println("Vector in original order using elementAt");
        for (int i=0;i<v.size();i++)
            System.out.print(v.elementAt(i)+" ");
        System.out.println();
        // Print out the original vector
        System.out.println("\nVector in reverse order using elementAt");
        for (int i=v.size()-1;i>=0;i++)
            System.out.print(v.elementAt(i)+" ");
```

```

    System.out.println();
    // Print out the original vector
    System.out.println("\nVector as a String");
    System.out.println(v.toString());
}
}

```

The output from this program looks like this:

Vector in original order using an Enumeration

0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19

Vector in original order using elementAt

0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19

Vector in reverse order using elementAt

19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0

Vector as a String

[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19]

Table The variables and methods available in the Vector interface.

Variable	Description
capacityIncrement	Size of the incremental allocations, in elements
elementCount	Number of elements in Vector
elementData	Buffer in which the elements are stored
Method	Description
	<b>Constructors</b>
Vector()	Constructs an empty vector
Vector(int)	Constructs an empty vector with the specified storage capacity
Vector(int, int)	Constructs an empty vector with the specified storage capacity and capacity increment
	<b>Methods</b>
addElement(Object)	Adds the specified object at the end of the

	Vector
capacity()	Returns the capacity of the Vector
clone()	Creates a clone of the Vector
contains(Object)	Returns true if the specified object is in the Vector
copyInto(Object[])	Copies the elements of this vector into an array
elementAt(int)	Returns the element at the specified index
elements()	Returns an Enumeration of the elements
ensureCapacity(int)	Ensures that the Vector has the specified capacity
firstElement()	Returns the first element of the Vector
indexOf(Object)	Returns the index of the first occurrence of the specified object within the Vector
indexOf(Object, int)	Returns the index of the specified object within the Vector, starting the search at the index specified and proceeding toward the end of the Vector
insertElementAt(Object, int)	Inserts an object at the index specified
isEmpty()	Returns true if the Vector is empty
lastElement()	Returns the last element of the Vector
lastIndexOf(Object)	Returns the index of the last occurrence of the specified object within the Vector
lastIndexOf(Object, int)	Returns the index of the specified object within the Vector, starting the search at the index specified and proceeding toward the beginning of the Vector
removeAllElements()	Removes all elements of the Vector
removeElement(Object)	Removes the specified object from the Vector

<code>removeElementAt(int)</code>	Removes the element with the specified index
<code>setElementAt(Object, int)</code>	Stores the object at the specified index in the Vector
<code>setSize(int)</code>	Sets the size of the Vector
<code>size()</code>	Returns the number of elements in the Vector
<code>toString()</code>	Converts the Vector to a string
<code>trimToSize()</code>	Trims the Vector's capacity down to the specified size

## Stack

The stack data structure is key to many programming efforts, ranging from building compilers to solving mazes. The Stack class in the Java library implements a Last In, First Out (LIFO) stack of objects. Even though they are based on (that is, they extend) the Vector class, Stack objects are typically not accessed in a direct fashion. Instead, values are pushed onto and popped off the top of the stack. The net effect is that the values most recently pushed are the first to pop.

The Stack1 code in the example pushes strings onto the stack and then retrieves them. The strings end up printed in the reverse order from which they were stored.

Listing Stack1.java: A sample Stack program.

```
import java.io.DataInputStream;
import java.io.BufferedReader;
import java.io.InputStreamReader;
import java.util.Stack;
import java.util.StringTokenizer;
class Stack1 {
    public static void main(String args[])
        throws java.io.IOException
    {
```

```

    BufferedReader dis=new BufferedReader(new
InputStreamReader(System.in));
    System.out.println("Enter a sentence: ");
    String s=dis.readLine();
    StringTokenizer st=new StringTokenizer(s);
    Stack stack=new Stack();
    while (st.hasMoreTokens())
        stack.push(st.nextToken());
    while (!stack.empty())
        System.out.print((String)stack.pop()+" ");
    System.out.println();
}
}

```

The output from this program looks like this:

Enter a sentence:

The quick brown fox jumps over the lazy dog  
dog lazy the over jumps fox brown quick The

Even though Stack objects normally are not accessed in a direct fashion, it is possible to search the Stack for a specific value using the search() method. search() accepts an object to find and returns the distance from the top of the Stack where the object was found. It returns -1 if the object is not found.

The method peek() returns the top object on the Stack without actually removing it from the Stack. The peek() method throws an EmptyStackException if the stack has no items.

Table summarizes the complete interface of the Stack class.

Table 11.12. The methods available in the Stack interface.

Method	Description
	Constructor
Stack()	Constructs an empty Stack
	Methods
empty()	Returns true if the Stack is empty

peek()	Returns the top object on the Stack without removing the element
pop()	Pops an element off the Stack
push(Object)	Pushes an element onto the Stack
search(Object)	Finds an object on the Stack

## Dictionary

The Dictionary class is an abstract class used as a base for the Hashtable class. It implements a data structure that allows a collection of key and value pairs to be stored. Any type of object can be used for the keys or the values. Typically, the keys are used to find a particular corresponding value.

Because the Dictionary class is an abstract class that cannot be used directly, the code examples presented in this section cannot actually be run. They are presented only to explain the purpose and use of the methods declared by this class. The following code would, hypothetically, be used to create a Dictionary with these values:

```
Dictionary products = new Dictionary();
products.put(new Integer(342), "Widget");
products.put(new Integer(124), "Gadget");
products.put(new Integer(754), "FooBar");
```

The put() method is used to insert a key and value pair into the Dictionary. Both arguments must be derived from the class Object. The key is the first argument and the value is the second argument.

A value can be retrieved using the get() method and a specific key to be found. get() returns the null value if the specified key is not found. Here's an example:

```
String name = products.get(new Integer(124));
if (name != null) {
    System.out.println("Product name for code 124 is " + name);
}
```

Although an individual object can be retrieved with the `get()` method, it is sometimes necessary to access all the keys or all the values. Two methods, `keys()` and `elements()`, return Enumerations that can be used to access the keys and the values.

Table summarizes the complete interface of the Dictionary class.

Table The methods available in the Dictionary interface.

Method	Description
	Constructor
<code>Dictionary()</code>	Constructs an empty Dictionary
	Methods
<code>elements()</code>	Returns an Enumeration of the values
<code>get(Object)</code>	Returns the object associated with the specified key
<code>isEmpty()</code>	Returns true if the Dictionary has no elements
<code>keys()</code>	Returns an Enumeration of the keys
<code>put(Object, Object)</code>	Stores the specified key and value pair in the Dictionary
<code>remove(Object)</code>	Removes an element from the Dictionary based on its key
<code>size()</code>	Returns the number of elements stored

## Hashtable

The hash table data structure is very useful when searching for and manipulating data. You should use the Hashtable class if you will be



storing a large amount of data in memory and then searching it. The time needed to complete a search of a hash table is decidedly less than what it takes to search a Vector. Of course, for small amounts of data, it doesn't make much difference whether you use a hash table or a linear data structure, because the overhead time is much greater than any search time would be.

Hash table organization is based on keys, which are computed based on the data being stored. For example, if you want to insert a number of words into a hash table, you can base your key on the first letter of the word. When you come back later to search for a word, you can then compute the key for the item being sought. By using this key, search time is drastically reduced because the items are stored based on the value of their respective key.

The Hashtable class implements a hash table storage mechanism for storing key and value pairs. Hash tables are designed to quickly locate and retrieve stored information by using a key. Keys and values can be of any object type, but the key object's class must implement the hashCode() and equals() methods.

Table The methods available in the Hashtable interface.

Method	Description
Constructors	
Hashtable()	Constructs an empty Hashtable
Hashtable(int)	Constructs an empty Hashtable with the specified capacity
Hashtable(int, float)	Constructs an empty Hashtable with the given capacity and load factor
Methods	
clear()	Deletes all elements from the Hashtable
clone()	Creates a clone of the Hashtable

<code>contains(Object)</code>	Returns true if the specified object is an element of the Hashtable
<code>containsKey(Object)</code>	Returns true if the Hashtable contains the specified key
<code>elements()</code>	Returns an Enumeration of the Hashtable's values
<code>get(Object)</code>	Returns the object associated with the specified key
<code>isEmpty()</code>	Returns true if the Hashtable has no elements
<code>keys()</code>	Returns an Enumeration of the keys
<code>put(Object, Object)</code>	Stores the specified key and value pair in the Hashtable
<code>rehash()</code>	Rehashes the contents of the table into a bigger table
<code>remove(Object)</code>	Removes an element from the Hashtable based on its key
<code>size()</code>	Returns the number of elements stored
<code>toString()</code>	Converts the contents to a very long string

## Properties

The Properties class is what enables end-users to customize their Java program. For example, you can easily store values such as foreground colors, background colors, font defaults, and so on and then have those values available to be reloaded. This arrangement is most useful for Java applications, but you can also implement it for applets. If you have an applet that is regularly used by multiple users, you can keep a properties file on your server for each different user; the properties file is accessed each time that user loads the applet.

The Properties class is a Hashtable, which can be repeatedly stored and restored from a stream. It is used to implement persistent properties. The Properties class also allows for an unlimited level of nesting, by

searching a default property list if the required property is not found. The fact that this class is an extension of the Hashtable class means that all methods available in the Hashtable class are also available in the Properties class.

The sample program Properties1 in Listing 11.11 creates two properties lists. One is the default property list and the other is the user-defined property list. When the user property list is created, the default Properties object is passed. When the user property list is searched, if the key value is not found, the default Properties list is searched.

```
import java.io.DataInputStream;
import java.lang.Integer;
import java.util.Properties;
class Properties1 {
    public static void main(String args[])
        throws java.io.IOException
    {
        int numElements=4;
        String defaultNames[]={"Red","Green","Blue","Purple"};
        int defaultValues[]={1,2,3,4};
        String userNames[]={"Red","Yellow","Orange","Blue"};
        int userValues[]={100,200,300,400};
        DataInputStream dis=new DataInputStream(System.in);
        Properties defaultProps=new Properties();
        Properties userProps=new Properties(defaultProps);
        for (int i=0;i<numElements;i++){
            defaultProps.put(defaultNames[i],
                Integer.toString(defaultValues[i]));
            userProps.put(userNames[i],
                Integer.toString(userValues[i]));
        }
        System.out.println("Default Properties");
        defaultProps.list(System.out);
        System.out.println("\nUser Defined Properties");
    }
}
```

```

userProps.list(System.out);
String keyValue;
System.out.println("\nWhich property to find? ");
keyValue=dis.readLine();
System.out.println("Property '"+keyValue+"' is '"+
    userProps.getProperty(keyValue)+"");
}
}

```

Notice that the `getProperties()` method is used instead of the inherited `get()` method. The `get()` method searches only the current `Properties` object. The `getProperties()` method must be used to search the default `Properties` list. An alternative form of the `getProperties()` method has a second argument: a `Properties` list that is to be searched instead of the default specified when the `Properties` object was created.

The `propertyNames()` method can be used to return an `Enumeration`, which can be used to index all the property names. This `Enumeration` includes the property names from the default `Properties` list. Likewise, the `list()` method, which prints the `Properties` list to the standard output, lists all the properties of the current `Properties` object and those in the default `Properties` object.

`Properties` objects can be written to and read from a stream using the `save()` and `load()` methods. In addition to the output or input stream, the `save()` method has an additional string argument that is written at the beginning of the stream as a header comment.

Table The variables and methods available in the `Properties` interface.

Variable	Description
<code>defaults</code>	Default <code>Properties</code> list to search
	<b>Constructors</b>
<code>Properties()</code>	Constructs an empty property list
<code>Properties(Properties)</code>	Constructs an empty property list with the specified default

	Methods
<code>getProperty(String)</code>	Returns a property given the key
<code>getProperty(String, String)</code>	Returns a property given the specified key and default
<code>list(PrintStream)</code>	Lists the properties to a stream for debugging
<code>load(InputStream)</code>	Reads the properties from an InputStream
<code>propertyNames()</code>	Returns an Enumeration all the keys
<code>save(OutputStream, String)</code>	Writes the properties to an OutputStream

## StringTokenizer

This section describes the function of the StringTokenizer class, which also could have been appropriately grouped with the other classes in Chapter 12, "The I/O Package," because it is so vital to the input and output functions demonstrated in that chapter. The StringTokenizer class enables you to parse a string into a number of smaller strings called tokens. This class works specifically for what is called "delimited text," which means that each individual substring of the string is separated by a delimiter. The delimiter can be anything ranging from an \* to YabaDaba. You simply specify what you want the class to look for when tokenizing the string.

The delimiter set can be specified when the StringTokenizer object is created, or it can be specified on a per-token basis. The default delimiter set is the set of whitespace characters. With this delimiter set, the class would find all the separate words in a string and tokenize them. For example, the StringTokenizer1 code in Listing prints out each word of the string on a separate line.

A sample StringTokenizer program.

```
import java.io.DataInputStream;
import java.io.BufferedReader;
```

```
import java.io.InputStreamReader;
import java.util.StringTokenizer;
class StringTokenizer1 {
    public static void main(String args[])
        throws java.io.IOException
    {
        BufferedReader dis=new BufferedReader(new
InputStreamReader(System.in));
        System.out.println("Enter a sentence: ");
        String s=dis.readLine();
        StringTokenizer st=new StringTokenizer(s);
        while (st.hasMoreTokens())
            System.out.println(st.nextToken());
    }
}
```

Here is the output from this listing:

```
Enter a sentence:
Four score and seven
Four
score
and
seven
```

Pure excitement. The method `countTokens()` returns the number of tokens remaining in the string using the current delimiter set--that is, the number of times `nextToken()` can be called before generating an exception. This is an efficient method because it does not actually construct the substrings that `nextToken()` must generate.

In addition to extending the `java.lang.Object` class, the `StringTokenizer` class implements the `java.util.Enumeration` interface.

Table summarizes the methods of the `StringTokenizer` class.

Table The methods available in the StringTokenizer interface.

Method	Description
	<b>Constructors</b>
StringTokenizer (String)	Constructs a StringTokenizer given a string using whitespace as delimiters
StringTokenizer (String, String)	Constructs a StringTokenizer given a string and a delimiter set
StringTokenizer (String, String, boolean)	Constructs a StringTokenizer given a string and a delimiter set; the final parameter is a boolean value which, if true, says that the delimiters must be returned as tokens (if this parameter is false, the tokens are not returned)
	<b>Methods</b>
countTokens()	Returns the number of tokens remaining in the string
hasMoreTokens()	Returns true if more tokens exist
nextToken()	Returns the next token of the string
nextToken(String)	Returns the next token, given a new delimiter set
hasMoreElements()	Returns true if more elements exist in the enumeration
nextElement()	Returns the next element of the enumeration using the current delimiter set

## Date

Before Java 1.1, the Date class was an extremely important class for handling dates and times. However, it was weak in some areas such as internationalization and dealing with the differences in daylight saving time in different locations.

With Java 1.1, the Date class has been relegated to just storing the exact time. Most of the old Date class methods have been deprecated and should no longer be used. All the methods for converting time between binary and human-readable form are now handled by the Calendar, GregorianCalendar, TimeZone, SimpleTimeZone, and Locale classes. GregorianCalendar

The default constructor is used when the current date and time are required. The other constructor takes a millisecond representation of time and creates a Date object based on it.

Table Useable methods available in the Date interface.

Method	Description
Constructors	
Date()	Constructs a date using today's date and time
Date(long)	Constructs a date using a single UTC value
Methods	
after(Date)	Returns true if the date is later than the specified date
before(Date)	Returns true if the date is earlier than the specified date
equals(Object)	Returns true if the date and the specified date are equal
getTime()	Returns the time as a single UTC value
hashCode()	Computes a hash code for the date
setTime(long)	Sets the time using a single UTC value
toString()	Converts a date to text using UNIX ctime() conventions

You will also find the before() and after() functions useful. They enable you to send in another instance of the Date class and compare that date to the value in the calling instance



## Calendar

The Calendar class is an abstract class used to convert dates. You can use this class to convert a Date object to fields, such as YEAR, MONTH, HOUR, and so on. You can also use these fields to update a Date object.

In the API definition, only one subclass of Calendar exists: GregorianCalendar. Because most people and virtually all businesses in the world use the Gregorian calendar,

Table 11.3 summarizes the methods available in the Calendar class.

Table 11.3. Methods in the Calendar class.

Method	Description
	Constructors
Calendar()	Creates a calendar with the default TimeZone and Locale
Calendar(TimeZone,Locale)	Creates a calendar with the given TimeZone and Locale
	Static Methods
getDefault()	Returns a calendar with the default TimeZone and Locale
getDefault(TimeZone)	Returns a calendar with the default Locale and given TimeZone
getDefault(Locale)	Returns a calendar with the given Locale and default TimeZone
getDefault(TimeZone,Locale)	Returns a calendar with the given TimeZone and Locale
getAvailableLocales()	Returns an array of all available locales
	Methods

getTime()	Returns the date and time
setTime(Date)	Sets the date and time
get(byte)	Returns the specified field
set(byte,int)	Sets the specified field to the specified value
set(int,int,int)	Sets the year, month, and date
set(int,int,int,int,int)	Sets the year, month, date, hour, and minute
set(int,int,int,int,int,int)	Sets the year, month, date, hour, minute, and second
clear()	Clears all fields
clear(byte)	Clears the specified field
isSet(int)	Returns true if the specified field is set
equals(Object)	Returns true if two objects are the same
before(Object)	Returns true if this object is before the given object
after(Object)	Returns true if this object is after the given object
add(byte,int)	Adds the given value to the field
roll(byte,boolean)	Increments or decrements (depending on the boolean value) the specified field by one unit
setTimeZone(TimeZone)	Sets the time zone this object is in
setValidationMode(boolean)	If the boolean value is set to true, invalid dates are allowed
getValidationMode()	Returns whether or not invalid dates are allowed
setFirstDayOfWeek(byte)	Sets the first day of the week

<code>getFirstDayOfWeek()</code>	Returns the first day of the week
<code>setMinimumDaysInFirstWeek(byte)</code>	Sets how many days are required to define the first week of the month
<code>getMinimumDaysInFirstWeek()</code>	Returns how many days are required to define the first week of the month
<code>getMinimum(byte)</code>	Returns the minimum possible value for the given field
<code>getMaximum(byte)</code>	Returns the maximum possible value for the given field
<code>getGreatestMinimum(byte)</code>	Returns the greatest possible minimum value for the field
<code>getLeastMaximum(byte)</code>	Returns the least possible maximum value for the given field
<code>Clone()</code>	Makes a copy of this object

## Random

For the programming of games and many other program types, it is important to be able to generate random numbers. Java includes the capability to generate random numbers efficiently and effectively.

The Random class implements a pseudo-random number data type that generates a stream of seemingly random numbers. To create a sequence of different pseudo-random values each time the application is run, create the Random object as follows:

```
Random r=new Random();
```

This statement seeds the random generator with the current time. On the other hand, consider the following statement:

```
Random r=new Random(326); // Pick any value
```

This statement seeds the random generator with the same value each time, resulting in the same sequence of pseudo-random numbers each time the application runs. The generator can be reseeded at any time using the `setSeed()` method.

Pseudo-random numbers can be generated using one of these functions: `nextInt()`, `nextLong()`, `nextFloat()`, `nextDouble()`, or `nextGaussian()`. The first four functions return integers, longs, floats, and doubles

Table summarizes the complete interface of the `Random` class.

Table :: The methods available in the `Random` interface.

Method	Description
Constructors	
<code>Random()</code>	Creates a new random number generator
<code>Random(long)</code>	Creates a new random number generator using a seed
Methods	
<code>nextDouble()</code>	Returns a pseudo-random, uniformly distributed double
<code>nextFloat()</code>	Returns a pseudo-random, uniformly distributed float
<code>nextGaussian()</code>	Returns a pseudo-random, Gaussian distributed double
<code>nextInt()</code>	Returns a pseudo-random, uniformly distributed integer
<code>nextLong()</code>	Returns a pseudo-random, uniformly distributed long
<code>setSeed(long)</code>	Sets the seed of the pseudo-random number generator

## Multithreading

### Basics

A thread--sometimes called an *execution context* or a *lightweight process*--is a single sequential flow of control within a program. Threads can be used to isolate tasks. Each

thread is a sequential flow of control within the same program.

## Customizing a Thread's run Method

The run method gives a thread something to do. Its code implements the thread's running behavior. It can do anything that can be encoded in Java statements: compute a list of prime's, sort some data, perform some animation.

The Thread class implements a generic thread that, by default, does nothing. That is, the implementation of its run method is empty. This is not particularly useful, so the Thread class defines API that lets a Runnable object provide a more interesting run method for a thread.

There are two techniques for providing a run method for a thread:

## Subclassing Thread and Overriding run

The first way to customize what a thread does when it is running is to subclass Thread (itself a Runnable object) and override its empty run method so that it does something. Let's look at the SimpleThread class, the first of two classes in this example, which does just that:

```
public class SimpleThread extends Thread {
    public SimpleThread(String str) {
        super(str);
    }
    public void run() {
        for (int i = 0; i < 10; i++) {
            System.out.println(i + " " + getName());
            try {
                sleep((long)(Math.random() * 1000));
            }
        }
    }
}
```

```

        } catch (InterruptedException e) {}
    }
    System.out.println("DONE! " + getName());
}
}

```

The first method in the SimpleThread class is a constructor that takes a String as its only argument. This constructor is implemented by calling a superclass constructor and is interesting to us only because it sets the Thread's name, which is used later in the program.

method is the heart of any Thread and where the action of the Thread takes place. The run method of the SimpleThread class contains a for loop that iterates ten times. In each iteration the method displays the iteration number and the name of the Thread, then sleeps for a random interval of up to 1 second. After the loop has finished, the run method prints DONE! along with the name of the thread. That's it for the SimpleThread class.

The TwoThreadsDemo class provides a main method that creates two SimpleThread threads: one is named "Jamaica" and the other is named "Fiji". (If you can't decide on where to go for vacation you can use this program to help you decide-- go to the island whose thread prints "DONE!" first.)

```

public class TwoThreadsDemo {
    public static void main (String[] args) {
        new SimpleThread("Jamaica").start();
        new SimpleThread("Fiji").start();
    }
}

```

The main method also starts each thread immediately following its construction by calling the start method. To save you from typing in this program, click [here](#) for the source code to the SimpleThread class and [here](#) for the source code to the TwoThreadsDemo program. Compile and run the program and watch your vacation fate unfold. You should see output similar to the following:

```
0 Jamaica
0 Fiji
1 Fiji
1 Jamaica
2 Jamaica
2 Fiji
3 Fiji
3 Jamaica
4 Jamaica
4 Fiji
5 Jamaica
5 Fiji
6 Fiji
6 Jamaica
7 Jamaica
7 Fiji
8 Fiji
9 Fiji
8 Jamaica
DONE! Fiji
9 Jamaica
DONE! Jamaica
```

Notice how the output from each thread is intermingled with the output from the other. This

is because both SimpleThread threads are running concurrently. Thus, both run methods are running at the same time and each thread is displaying its output at the same time as the other.

Now, let's look at another example, the Clock applet, that uses the other technique for providing a run method to a Thread.

### Implementing the Runnable Interface

The Clock applet shown below displays the current time and updates its display every second. You can scroll this page and perform other tasks while the clock continues to update because the code that updates the clock's display runs within its own thread.

The Clock applet uses a different technique than SimpleThread for providing the run method for its thread. Instead of subclassing Thread, Clock implements the Runnable interface (and therefore implements the run method defined in it). Clock then creates a thread and provides itself as an argument to the Thread's constructor. When created in this way, the Thread gets its run method from the object passed into the constructor. The code that accomplishes this is shown in bold here:

```
import java.awt.Graphics;
import java.util.*;
import java.text.DateFormat;
import java.applet.Applet;

public class Clock extends Applet implements Runnable {
    private Thread clockThread = null;
    public void start() {
        if (clockThread == null) {
            clockThread = new Thread(this, "Clock");
            clockThread.start();
        }
    }
    public void run() {
```



```

Thread myThread = Thread.currentThread();
while (clockThread == myThread) {
    repaint();
    try {
        Thread.sleep(1000);
    } catch (InterruptedException e){
        // the VM doesn't want us to sleep anymore,
        // so get back to work
    }
}
}
public void paint(Graphics g) {
    // get the time and convert it to a date
    Calendar cal = Calendar.getInstance();
    Date date = cal.getTime();
    // format it and display it
    DateFormat dateFormatter = DateFormat.getTimeInstance();
    g.drawString(dateFormatter.format(date), 5, 10);
}
// overrides Applet's stop method, not Thread's
public void stop() {
    clockThread = null;
}
}

```

The Clock applet's run method loops until the browser asks it to stop. During each iteration of the loop, the clock repaints its display. The paint method figures out what time it is, formats it in a localized way, and displays it.

## Deciding to Use the Runnable Interface

You have now seen two ways to provide the run method for a Java thread:

1. Subclass the Thread class defined in the java.lang package and override the run method.
2. Provide a class that implements the Runnable interface (also defined in the java.lang package) and therefore implements the run method. In this case, a Runnable object provides the run method to the thread.

There are good reasons for choosing either of these options over the other. However, for most cases, including that of the Clock applet, the following rule of thumb will guide you to the best option.

**Rule of Thumb:** If your class must subclass some other class (the most common example being Applet), you should use Runnable as described in option #2.

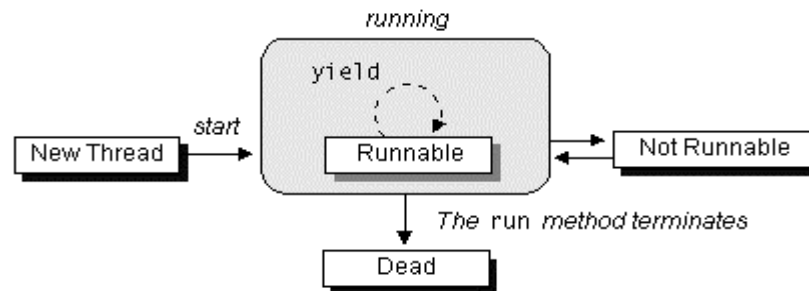
To run in a Java-enabled browser, the Clock class has to be a subclass of the Applet class. Also, the Clock applet needs a thread so that it can continuously update its display without taking over the process in which it is running. But since the Java language does not support multiple class inheritance, the Clock class cannot be a subclass of both Thread and Applet. Thus the Clock class must use the Runnable interface to provide its threaded behavior.

## The Life Cycle of a Thread

Now that you've seen how to give a thread something to do, we'll review some details that were glossed over in the previous section. In particular, we look at the life cycle of a thread: how to create and start a thread, some of the special things it can do while it's running, and how to stop it.

The following diagram shows the states that a Java thread can be in during its life. It also illustrates which method calls cause a transition to another state. This figure is not a

complete finite state diagram, but rather an overview of the more interesting and common facets of a thread's life. The remainder of this section uses the Clock applet previously introduced to discuss a thread's life cycle in terms of its state.



## Creating a Thread

The application in which an applet is running calls the applet's start method when the user visits the applet's page. The Clock applet creates a Thread, `clockThread`, in its start with the bold code shown here:

```

public void start() {
    if (clockThread == null) {
        clockThread = new Thread(this, "Clock");
        clockThread.start();
    }
}
  
```

After the bold statement has been executed, `clockThread` is in the New Thread state. When a thread is a New Thread, it is merely an empty Thread object; no system resources have been allocated for it yet. When a thread is in this state, you can only start the thread. Calling any method besides start when a thread is in this state makes no sense and causes an `IllegalThreadStateException`. (In fact, the runtime system throws an `IllegalThreadStateException` any time a method is called on a thread and that thread's state does not allow for that method call.)

Notice that `this`--the Clock instance-- is the first argument to the thread constructor. The first argument to this thread constructor must implement the `Runnable` interface and provides the thread with its run method. The second argument is just a name for the thread.

## Starting a Thread

Now consider the next line of code in Clock's start method shown here in bold:

```
public void start() {  
    if (clockThread == null) {  
        clockThread = new Thread(this, "Clock");  
        clockThread.start();  
    }  
}
```

The start method creates the system resources necessary to run the thread, schedules the thread to run, and calls the thread's run method. clockThread's run method is the one defined in the Clock class.

After the start method has returned, the thread is "running". Yet, it's somewhat more complex than that. As the previous figure shows, a thread that has been started is actually in the Runnable state. Many computers have a single processor, thus making it impossible to run all "running" threads at the same time. The Java runtime system must implement a scheduling scheme that shares the processor between all "running" threads. So at any given time, a "running" thread actually may be waiting for its turn in the CPU.

### Making a Thread Not Runnable

A thread becomes Not Runnable when one of these events occurs:

- Its sleep method is invoked.
- The thread calls the wait method to wait for a specific condition to be satisfied.
- The thread is blocking on I/O.

The clockThread in the Clock applet becomes Not Runnable when the run method calls sleep on the current thread:

```
public void run() {
    Thread myThread = Thread.currentThread();
    while (clockThread == myThread) {
        repaint();
        try {
            Thread.sleep(1000);
        } catch (InterruptedException e) {
            // the VM doesn't want us to sleep anymore,
            // so get back to work
        }
    }
}
```

During the second that the clockThread is asleep, the thread does not run, even if the processor becomes available. After the second has elapsed, the thread becomes Runnable again and, if the processor becomes available, the thread begins running again.

### Stopping a Thread

A program doesn't stop a thread like it stops an applet (by calling a method). Rather, a thread arranges for its own death by having a run method that terminates naturally. For example, the while loop in this run method is a finite loop-- it will iterate 100 times and then exit:

```
public void run() {
    int i = 0;
    while (i < 100) {
        i++;
        System.out.println("i = " + i);
    }
}
```

A thread with this run method dies naturally when the loop completes and the run method exits.

Let's look at how the Clock applet thread arranges for its own death. You might want to use this technique with your applets. Recall Clock's run method:

```
public void run() {
    Thread myThread = Thread.currentThread();
    while (clockThread == myThread) {
        repaint();
        try {
            Thread.sleep(1000);
        } catch (InterruptedException e) {
            // the VM doesn't want us to sleep anymore,
            // so get back to work
        }
    }
}
```

The exit condition for this run method is the exit condition for the while loop because there is no code after the while loop:

```
while (clockThread == myThread) {
```

This condition indicates that the loop will exit when the currently executing thread is not equal to clockThread. When would this ever be the case?

When you leave the page, the application in which the applet is running calls the applet's stop method. This method then sets the clockThread to null, thereby telling the main loop in the run method to terminate:

```
public void stop() { // applets' stop method
    clockThread = null;
}
```

If you revisit the page, the start method is called again and the clock starts up again with a new thread. Even if you stop and start the applet

faster than one iteration of the loop, `clockThread` will be a different thread than `myThread` and the loop will still terminate.

## The `isAlive` Method

A final word about thread state: The API for the `Thread` class includes a method called `isAlive`. The `isAlive` method returns true if the thread has been started and not stopped. If the `isAlive` method returns false, you know that the thread either is a New Thread or is Dead. If the `isAlive` method returns true, you know that the thread is either Runnable or Not Runnable. You cannot differentiate between a New Thread or a Dead thread. Nor can you differentiate between a Runnable thread and a Not Runnable thread.

## Understanding Thread Priority

Previously, this lesson claimed that threads run concurrently. While conceptually this is true, in practice it usually isn't. Most computer configurations have a single CPU, so threads actually run one at a time in such a way as to provide an illusion of concurrency. Execution of multiple threads on a single CPU, in some order, is called *scheduling*. The Java runtime supports a very simple, deterministic scheduling algorithm known as *fixed priority scheduling*. This algorithm schedules threads based on their *priority* relative to other runnable threads.

When a Java thread is created, it inherits its priority from the thread that created it. You can also modify a thread's priority at any time after its creation using the `setPriority` method. Thread priorities are integers ranging between `MIN_PRIORITY` and `MAX_PRIORITY` (constants defined in the `Thread` class). The higher the integer, the higher the

priority. At any given time, when multiple threads are ready to be executed, the runtime system chooses the runnable thread with the highest priority for execution. Only when that thread stops, yields, or becomes not runnable for some reason will a lower priority thread start executing. If two threads of the same priority are waiting for the CPU, the scheduler chooses one of them to run in a round-robin fashion. The chosen thread will run until one of the following conditions is true:

- A higher priority thread becomes runnable.
- It yields, or its `run` method exits.
- On systems that support time-slicing, its time allotment has expired.

Then the second thread is given a chance to run, and so on, until the interpreter exits.

The Java runtime system's thread scheduling algorithm is also *preemptive*. If at any time a thread with a higher priority than all other runnable threads becomes runnable, the runtime system chooses the new higher priority thread for execution. The new higher priority thread is said to *preempt* the other threads.

**Rule of thumb:** At any given time, the highest priority thread is running. However, this is not guaranteed. The thread scheduler may choose to run a lower priority thread to avoid starvation. For this reason, use priority only to affect scheduling policy for efficiency purposes. Do not rely on thread priority for algorithm correctness.

## Time-Slicing

Some systems, such as Windows 95/NT, fight selfish thread behavior with a strategy known as *time-slicing*. Time-slicing



comes into play when there are multiple "Runnable" threads of equal priority and those threads are the highest priority threads competing for the CPU. For example,

```
public void run() {
    while (tick < 400000) {
        tick++;
        if ((tick % 50000) == 0)
            System.out.println("Thread #" + num
                + ", tick = " + tick);
    }
}
```

This run contains a tight loop that increments the integer tick and every 50,000 ticks prints out the thread's identifier and its tick count.

When running this program on a time-sliced system, you will see messages from both threads intermingled with one another. Like this:

```
Thread #1, tick = 50000
Thread #0, tick = 50000
Thread #0, tick = 100000
Thread #1, tick = 100000
Thread #1, tick = 150000
Thread #1, tick = 200000
Thread #0, tick = 150000
Thread #0, tick = 200000
Thread #1, tick = 250000
Thread #0, tick = 250000
Thread #0, tick = 300000
Thread #1, tick = 300000
Thread #1, tick = 350000
Thread #0, tick = 350000
Thread #0, tick = 400000
Thread #1, tick = 400000
```

This output is produced because a time-sliced system divides the CPU into time slots and iteratively gives each of the equal-and-highest priority threads a time slot in which to run. The time-sliced system

iterates through the equal-and-highest priority threads, allowing each one a bit of time to run, until one or more of them finishes or until a higher priority thread preempts them. Notice that time-slicing makes no guarantees as to how often or in what order threads are scheduled to run. When running this program on a non-time-sliced system, however, you will see messages from one thread finish printing before the other thread ever gets a chance to print one message. Like this:

```
Thread #0, tick = 50000
Thread #0, tick = 100000
Thread #0, tick = 150000
Thread #0, tick = 200000
Thread #0, tick = 250000
Thread #0, tick = 300000
Thread #0, tick = 350000
Thread #0, tick = 400000
Thread #1, tick = 50000
Thread #1, tick = 100000
Thread #1, tick = 150000
Thread #1, tick = 200000
Thread #1, tick = 250000
Thread #1, tick = 300000
Thread #1, tick = 350000
Thread #1, tick = 400000
```

This is because a non-time-sliced system chooses one of the equal-and-highest priority threads to run and allows that thread to run until it relinquishes the CPU (by sleeping, yielding, finishing its job) or until a higher priority preempts it.

**Note:** The Java runtime does not implement (and therefore does not guarantee) time-slicing. However, some systems on which you can run Java do support time-slicing. Your Java programs should not rely on time-slicing as it may produce different results on different systems.

## Summary

- Most computers have only one CPU, so threads must share the CPU with other threads. The execution of multiple threads on a single CPU, in some order, is called scheduling. The Java runtime supports a very simple, deterministic scheduling algorithm known as fixed priority scheduling.
- Each Java thread is given a numeric priority between `MIN_PRIORITY` and `MAX_PRIORITY` (constants defined in the `Thread` class). At any given time, when multiple threads are ready to be executed, the thread with the highest priority is chosen for execution. Only when that thread stops, or is suspended for some reason, will a lower priority thread start executing.
- Scheduling of the CPU is fully preemptive. If a thread with a higher priority than the currently executing thread needs to execute, the higher priority thread is immediately scheduled.
- The Java runtime will not preempt the currently running thread for another thread of the same priority. In other words, the Java runtime does not time-slice. However, the system implementation of threads underlying the Java `Thread` class may support time-slicing. Do not write code that relies on time-slicing.
- In addition, a given thread may, at any time, give up its right to execute by calling the `yield` method. Threads can only yield the CPU to other threads of the same priority--attempts to yield to a lower priority thread are ignored.
- When all the runnable threads in the system have the same priority, the scheduler chooses the next thread to run in a simple, non-preemptive, round-robin scheduling order.

## Synchronizing Threads

So far, this lesson has contained examples with independent, asynchronous threads. That is, each thread contained all of the data and methods required for its execution and didn't

require any outside resources or methods. In addition, the threads in those examples ran at their own pace without concern over the state or activities of any other concurrently running threads.

However, there are many interesting situations where separate, concurrently running threads do share data and must consider the state and activities of other threads. One such set of programming situations are known as producer/consumer scenarios where the producer generates a stream of data which then is consumed by a consumer.

For example, imagine a Java application where one thread (the producer) writes data to a file while a second thread (the consumer) reads data from the same file. Or, as you type characters on the keyboard, the producer thread places key events in an event queue and the consumer thread reads the events from the same queue. Both of these examples use concurrent threads that share a common resource: the first shares a file, the second shares an event queue. Because the threads share a common resource, they must be synchronized in some way.

### **The Producer/Consumer Example**

The Producer generates an integer between 0 and 9 (inclusive), stores it in a CubbyHole object, and prints the generated number. To make the synchronization problem more interesting, the Producer sleeps for a random amount of time between 0 and 100 milliseconds before repeating the number generating cycle:

```
public class Producer extends Thread {
    private CubbyHole cubbyhole;
    private int number;
```

```

public Producer(CubbyHole c, int number) {
    cubbyhole = c;
    this.number = number;
}

public void run() {
    for (int i = 0; i < 10; i++) {
        cubbyhole.put(i);
        System.out.println("Producer #" +
this.number
        + " put: " + i);
        try {
            sleep((int)(Math.random() * 100));
        } catch (InterruptedException e) { }
    }
}
}

```

The Consumer, being ravenous, consumes all integers from the CubbyHole (the exact same object into which the Producer put the integers in the first place) as quickly as they become available.

```

public class Consumer extends Thread {
    private CubbyHole cubbyhole;
    private int number;

    public Consumer(CubbyHole c, int number) {
        cubbyhole = c;
        this.number = number;
    }

    public void run() {

```

```

        int value = 0;
        for (int i = 0; i < 10; i++) {
            value = cubbyhole.get();
            System.out.println("Consumer #" +
this.number
                                + " got: " + value);
        }
    }
}

```

The Producer and Consumer in this example share data through a common CubbyHole object. And you will note that neither the Producer nor the Consumer makes any effort whatsoever to ensure that the Consumer is getting each value produced once and only once. The synchronization between these two threads actually occurs at a lower level, within the get and put methods of the CubbyHole object. However, let's assume for a moment that these two threads make no arrangements for synchronization and talk about the potential problems that might arise in that situation.

One problem arises when the Producer is quicker than the Consumer and generates two numbers before the Consumer has a chance to consume the first one. Thus the Consumer would skip a number. Part of the output might look like this:

...

```
Consumer #1 got: 3
Producer #1 put: 4
Producer #1 put: 5
Consumer #1 got: 5
```

...

Another problem that might arise is when the Consumer is quicker than the Producer and consumes the same value twice. In this situation, the Consumer would print the same value twice and might produce output that looked like this:

```
...
Producer #1 put: 4
Consumer #1 got: 4
Consumer #1 got: 4
Producer #1 put: 5
```

...

Either way, the result is wrong. You want the Consumer to get each integer produced by the Producer exactly once. Problems such as those just described are called *race conditions*. They arise from multiple, asynchronously executing threads trying to access a single object at the same time and getting the wrong result.

Race conditions in the producer/consumer example are prevented by having the storage of a new integer into the

CubbyHole by the Producer be synchronized with the retrieval of an integer from the CubbyHole by the Consumer. The Consumer must consume each integer exactly once.

The activities of the Producer and Consumer must be synchronized in two ways. First, the two threads must not simultaneously access the CubbyHole. A Java thread can prevent this from happening by locking an object. When an object is locked by one thread and another thread tries to call a synchronized method on the same object, the second thread will block until the object is unlocked.

And second, the two threads must do some simple coordination. That is, the Producer must have some way to indicate to the Consumer that the value is ready and the Consumer must have some way to indicate that the value has been retrieved. The Thread class provides a collection of methods--wait, notify, and notifyAll--to help threads wait for a condition and notify other threads of when that condition changes.

## The Main Program

Here's a small stand-alone Java application that creates a CubbyHole object, a Producer, a Consumer, and then starts both the Producer and the Consumer.

```
public class ProducerConsumerTest {
    public static void main(String[] args) {
        CubbyHole c = new CubbyHole();
        Producer p1 = new Producer(c, 1);
        Consumer c1 = new Consumer(c, 1);

        p1.start();
        c1.start();
    }
}
```



```
}
```

## The Output

Here's the output of ProducerConsumerTest.

```
Producer #1 put: 0  
Consumer #1 got: 0  
Producer #1 put: 1  
Consumer #1 got: 1  
Producer #1 put: 2  
Consumer #1 got: 2  
Producer #1 put: 3  
Consumer #1 got: 3  
Producer #1 put: 4  
Consumer #1 got: 4  
Producer #1 put: 5  
Consumer #1 got: 5  
Producer #1 put: 6  
Consumer #1 got: 6  
Producer #1 put: 7  
Consumer #1 got: 7  
Producer #1 put: 8  
Consumer #1 got: 8  
Producer #1 put: 9  
Consumer #1 got: 9
```

## Locking an Object

The code segments within a program that access the same object from separate, concurrent threads are called *critical sections*. In the Java language, a critical section can be a block or a method and are identified with the synchronized keyword. The Java platform then associates a lock with every object that has synchronized code.

In the producer/consumer example, the put and get methods of the CubbyHole are the critical sections. The Consumer should not access the CubbyHole when the Producer is changing it, and the Producer should

not modify it when the Consumer is getting the value. So put and get in the CubbyHole class should be marked with the synchronized keyword. Here's a code skeleton for the CubbyHole class:

```
public class CubbyHole {
    private int contents;
    private boolean available = false;

    public synchronized int get() {
        ...
    }

    public synchronized void put(int value) {
        ...
    }
}
```

Note that the method declarations for both put and get contain the synchronized keyword. Hence, the system associates a unique lock with every instance of CubbyHole (including the one shared by the Producer and the Consumer). Whenever control enters a synchronized method, the thread that called the method locks the object whose method has been called. Other threads cannot call a synchronized method on the same object until the object is unlocked.

So, when the Producer calls CubbyHole's put method, it locks the CubbyHole, thereby preventing the Consumer from calling the CubbyHole's get method:

```
public synchronized void put(int value) {
    // CubbyHole locked by the Producer
    ..
    // CubbyHole unlocked by the Producer
}
```

When the put method returns, the Producer unlocks the CubbyHole. Similarly, when the Consumer calls CubbyHole's get method, it locks the CubbyHole, thereby preventing the Producer from calling put:

```
public synchronized int get() {
    // CubbyHole locked by the Consumer
```

```

...
// CubbyHole unlocked by the Consumer
}

```

The acquisition and release of a lock is done automatically and atomically by the Java runtime system. This ensures that race conditions cannot occur in the underlying implementation of the threads, thus ensuring data integrity. Synchronization isn't the whole story. The two threads must also be able to notify one another when they've done their job. Learn more about that after a brief foray into reentrant locks.

### Requiring a Lock

The Java runtime system allows a thread to re-acquire a lock that it already holds because Java locks are reentrant. Reentrant locks are important because they eliminate the possibility of a single thread deadlocking itself on a lock that it already holds.

Consider this class:

```

public class Reentrant {
    public synchronized void a() {
        b();
        System.out.println("here I am, in a()");
    }
    public synchronized void b() {
        System.out.println("here I am, in b()");
    }
}

```

Reentrant contains two synchronized methods: a and b. The first synchronized method, a, calls the other synchronized method, b. When control enters method a, the current thread acquires the lock for the Reentrant object. Now, a calls b and because b is also synchronized the thread attempts to acquire the same lock again. Because Java supports reentrant locks, this works. The current thread can acquire the Reentrant object's lock again and both a and b execute to conclusion as is evidenced by the output:

```

here I am, in b()
here I am, in a()

```

In systems that don't support reentrant locks, this sequence of method calls would cause deadlock.

### Using the notifyAll and wait Methods

The CubbyHole stores its value in a private member variable called contents. CubbyHole has another private member variable, available, that is a boolean. available is true when the value has just been put but not yet gotten and is false when the value has been gotten but not yet put. So, here's one possible implementation for the put and get methods:

```
public synchronized int get() { // won't work!
    if (available == true) {
        available = false;
        return contents;
    }
}
public synchronized void put(int value) { //
won't work!
    if (available == false) {
        available = true;
        contents = value;
    }
}
```

As implemented, these two methods won't work. Look at the get method. What happens if the Producer hasn't put anything in the CubbyHole and available isn't true? get does nothing. Similarly, if the Producer calls put before the Consumer got the value, put doesn't do anything.

You really want the Consumer to wait until the Producer puts something in the CubbyHole and the Producer must notify the Consumer when it's done so. Similarly, the Producer must wait until the Consumer takes a value (and notifies the Producer of its activities) before replacing it with a new value. The two threads must coordinate more fully and can use Object's wait and notifyAll methods to do so.

Here are the new implementations of get and put that wait on and notify each other of their activities:

```
public synchronized int get() {
    while (available == false) {
        try {
            // wait for Producer to put value
            wait();
        } catch (InterruptedException e) {
        }
    }
    available = false;
    // notify Producer that value has been retrieved
    notifyAll();
    return contents;
}

public synchronized void put(int value) {
    while (available == true) {
        try {
            // wait for Consumer to get value
            wait();
        } catch (InterruptedException e) {
        }
    }
    contents = value;
    available = true;
    // notify Consumer that value has been set
```

```
        notifyAll();  
    }
```

The code in the get method loops until the Producer has produced a new value. Each time through the loop, get calls the wait method. The wait method relinquishes the lock held by the Consumer on the CubbyHole (thereby allowing the Producer to get the lock and update the CubbyHole) and then waits for notification from the Producer. When the Producer puts something in the CubbyHole, it notifies the Consumer by calling notifyAll. The Consumer then comes out of the wait state, available is now true, the loop exits, and the get method returns the value in the CubbyHole.

The put method works in a similar fashion, waiting for the Consumer thread to consume the current value before allowing the Producer to produce a new one.

The notifyAll method wakes up all threads waiting on the object in question (in this case, the CubbyHole). The awakened threads compete for the lock. One thread gets it, and the others go back to waiting. The Object class also defines the notify method, which arbitrarily wakes up one of the threads waiting on this object.

The Object class contains not only the version of wait that is used in the producer/consumer example and which waits indefinitely for notification, but also two other versions of the wait method:

**wait(long timeout)**

Waits for notification or until the timeout period has elapsed. timeout is measured in milliseconds.

**wait(long timeout, int nanos)**

Waits for notification or until timeout milliseconds plus nanos nanoseconds have elapsed.

**Note:** Besides using these timed wait methods to synchronize threads, you also can use them in place of sleep. Both wait and sleep delay for the requested amount of time, but you can easily wake up wait with a notify but a sleeping thread cannot be awakened prematurely. This doesn't matter too much for threads that don't sleep for long, but it could be important for threads that sleep for minutes at a time.

## Avoiding Starvation and Deadlock

The dining philosophers are often used to illustrate various problems that can occur when many synchronized threads are competing for limited resources.

The story goes like this: Five philosophers are sitting at a round table. In front of each philosopher is a bowl of rice. Between each pair of philosophers is one chopstick. Before an individual philosopher can take a bite of rice he must have two chopsticks--one taken from the left, and one taken from the right. The philosophers must find some way to share chopsticks such that they all get to eat.

For most Java programmers, the best choice is to prevent deadlock rather than to try and detect it. Deadlock detection is complicated and beyond the scope of this tutorial. The simplest approach to preventing deadlock is to impose ordering on the condition variables. In the dining philosopher example, there is no ordering imposed on the condition

variables because the philosophers and the chopsticks are arranged in a circle.

However, we can change the rules in the applet by numbering the chopsticks 1 through 5 and insisting that the philosophers pick up the chopstick with the lower number first. The philosopher who is sitting between chopsticks 1 and 2 and the philosopher who is sitting between chopsticks 1 and 5 must now reach for the same chopstick first (chopstick 1) rather than picking up the one on the right. Whoever gets chopstick 1 first is now free to take another one. Whoever doesn't get chopstick 1 must now wait for the first philosopher to release it. Deadlock is not possible.

## Grouping Threads

Every Java thread is a member of a *thread group*. Thread groups provide a mechanism for collecting multiple threads into a single object and manipulating those threads all at once, rather than individually. For example, you can start or suspend all the threads within a group with a single method call. Java thread groups are implemented by the ThreadGroup class in the `java.lang` package.

The runtime system puts a thread into a thread group during thread construction. When you create a thread, you can either allow the runtime system to put the new thread in some reasonable default group or you can explicitly set the new thread's group. The thread is a permanent member of whatever thread group it joins upon its creation--you cannot move a thread to a new group after the thread has been created.



## The Default Thread Group

If you create a new Thread without specifying its group in the constructor, the runtime system automatically places the new thread in the same group as the thread that created it (known as the *current thread group* and the *current thread*, respectively). So, if you leave the thread group unspecified when you create your thread, what group contains your thread?

When a Java application first starts up, the Java runtime system creates a ThreadGroup named main. Unless specified otherwise, all new threads that you create become members of the main thread group.

**Note:** If you create a thread within an applet, the new thread's group may be something other than main, depending on the browser or viewer that the applet is running in.

### Creating a Thread Explicitly in a Group

As mentioned previously, a thread is a permanent member of whatever thread group it joins when its created--you cannot move a thread to a new group after the thread has been created. Thus, if you wish to put your new thread in a thread group other than the default, you must specify the thread group explicitly when you create the thread. The Thread class has three constructors that let you set a new thread's group:

```
public Thread(ThreadGroup group, Runnable runnable)
public Thread(ThreadGroup group, String name)
public Thread(ThreadGroup group, Runnable runnable, String name)
```

Each of these constructors creates a new thread, initializes it based on the Runnable and String parameters, and makes the new thread a member of the specified group. For example, the following code sample creates a thread group (myThreadGroup) and then creates a thread (myThread) in that group.

```
ThreadGroup myThreadGroup = new ThreadGroup(
```

```
        "My Group of Threads");  
Thread myThread = new Thread(myThreadGroup,  
        "a thread for my group");
```

### Getting a Thread's Group

To find out what group a thread is in, you can call its `getThreadGroup` method:

```
theGroup = myThread.getThreadGroup();
```

## The ThreadGroup Class

Once you've obtained a thread's `ThreadGroup`, you can query the group for information, such as what other threads are in the group. You can also modify the threads in that group, such as suspending, resuming, or stopping them, with a single method invocation.

## Collection Management Methods

The `ThreadGroup` provides a set of methods that manage the threads and subgroups within the group and allow other objects to query the `ThreadGroup` for information about its contents. For example, you can call `ThreadGroup`'s `activeCount` method to find out the number of active threads currently in the group. The `activeCount` method is often used with the `enumerate` method to get an array filled with references to all the active threads in a `ThreadGroup`. For example, the `listCurrentThreads` method in the following example fills an array with all of the active threads in the current thread group and prints their names:

```
public class EnumerateTest {  
    public void listCurrentThreads() {  
        ThreadGroup currentGroup =  
  
        Thread.currentThread().getThreadGroup();
```

```

        int numThreads =
currentGroup.activeCount();
        Thread[] listOfThreads = new
Thread[numThreads];

        currentGroup.enumerate(listOfThreads);
        for (int i = 0; i < numThreads; i++)
            System.out.println("Thread #" + i + " = " +
listOfThreads[i].getName());
    }
}

```

Other collection management methods provided by the ThreadGroup class include activeGroupCount and list.

## Methods that Operate on the Group

The ThreadGroup class supports several attributes that are set and retrieved from the group as a whole. These attributes include the maximum priority that any thread within the group can have, whether the group is a "daemon" group, the name of the group, and the parent of the group.

The methods that get and set ThreadGroup attributes operate at the group level. They inspect or change the attribute on the ThreadGroup object, but do not affect any of the threads within the group. The following is a list of ThreadGroup methods that operate at the group level:

- getMaxPriority and setMaxPriority
- getDaemon and setDaemon
- getName
- getParent and parentOf
- toString

For example, when you use `setMaxPriority` to change a group's maximum priority, you are only changing the attribute on the group object; you are not changing the priority of any of the threads within the group. Consider the following program that creates a group and a thread within that group:

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

        ThreadGroup groupNORM = new
ThreadGroup(
                "A group with normal
priority");
        Thread priorityMAX = new
Thread(groupNORM,
                "A thread with maximum
priority");

        // set Thread's priority to max (10)

priorityMAX.setPriority(Thread.MAX_PRIORIT
Y);

        // set ThreadGroup's max priority to normal
(5)

groupNORM.setMaxPriority(Thread.NORM_PRI
ORITY);

        System.out.println("Group's maximum
priority = " +

groupNORM.getMaxPriority());
        System.out.println("Thread's priority = " +
```

```
        priorityMAX.getPriority());  
    }  
}
```

When the ThreadGroup groupNORM is created, it inherits its maximum priority attribute from its parent thread group. In this case, the parent group priority is the maximum (MAX\_PRIORITY) allowed by the Java runtime system. Next the program sets the priority of the priorityMAX thread to the maximum allowed by the Java runtime system. Then the program lowers the group's maximum to the normal priority (NORM\_PRIORITY). The setMaxPriority method does not affect the priority of the priorityMAX thread, so that at this point, the priorityMAX thread has a priority of 10, which is greater than the maximum priority of its group, groupNORM. This is the output from the program:

Group's maximum priority = 5  
Thread's priority = 10

As you can see a thread can have a higher priority than the maximum allowed by its group as long as the thread's priority is set before the group's maximum priority is lowered. A thread group's maximum priority is used to limit a thread's

priority when the thread is first created within a group or when you use `setPriority` to change the thread's priority. Note that `setMaxPriority` *does* change the maximum priority of all of its descendant-threadgroups.

Similarly, a group's daemon status applies only to the group. Changing a group's daemon status does not affect the daemon status of any of the threads in the group. Furthermore, a group's daemon status does not in any way imply the daemon status of its threads--you can put any thread within a daemon thread group. The daemon status of a thread group simply indicates that the group will be destroyed when all of its threads have been terminated.

### **Methods that Operate on All Threads within a Group**

The `ThreadGroup` class has three methods that allow you to modify the current state of all the threads within that group:

- `resume`
- `stop`
- `suspend`

These methods apply the appropriate state change to every thread in the thread group and its subgroups.

### **Access Restriction Methods**

The `ThreadGroup` class itself does not impose any access restrictions, such as allowing threads from one group to inspect or modify threads in a different group. Rather the `Thread` and `ThreadGroup` classes cooperate with security managers (subclasses of the `SecurityManager` class), which

can impose access restrictions based on thread group membership.

The Thread and ThreadGroup class both have a method, checkAccess, which calls the current security manager's checkAccess method. The security manager decides whether to allow the access based on the group membership of the threads involved. If access is not allowed, the checkAccess method throws a SecurityException. Otherwise, checkAccess simply returns.

The following is a list of ThreadGroup methods that call ThreadGroup's checkAccess before performing the action of the method. These are what are known as *regulated accesses*, that is, accesses that must be approved by the security manager before they can be completed.

- ThreadGroup(ThreadGroup *parent*, String *name*)
- setDaemon(boolean *isDaemon*)
- setMaxPriority(int *maxPriority*)
- stop
- suspend
- resume
- destroy

This is a list of the methods in the Thread class that call checkAccess before proceeding:

- constructors that specify a thread group
- stop
- suspend
- resume
- setPriority(int *priority*)
- setName(String *name*)
- setDaemon(boolean *isDaemon*)

A stand-alone Java application does not have a security manager by default; no restrictions are imposed and any thread can inspect or modify any other thread, regardless of the group they are in. You can define and implement your own access restrictions for thread groups by subclassing **SecurityManager**

The HotJava Web browser is an example of an application that implements its own security manager. HotJava needs to ensure that applets are well-behaved and don't do nasty things to other applets running at the same time (such as lowering the priority of another applet's threads). HotJava's security manager does not allow threads in different groups to modify one another. Please note that access restrictions based on thread groups may vary from browser to browser and thus applets may behave differently in different browsers.

## **Summary**

This lesson has provided a great deal of information about using threads in the Java development environment. Threads are supported by various components of the Java development environment, and it can be hard to find the features that you need. This section summarizes where in the Java environment you can find various classes, methods, and language features that participate in the Java

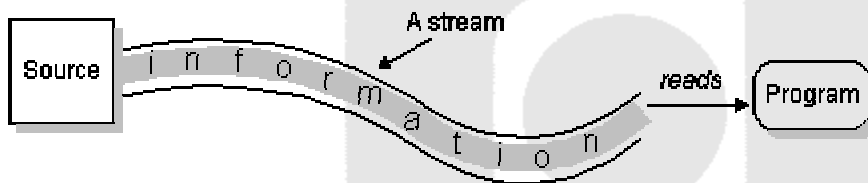


## IO

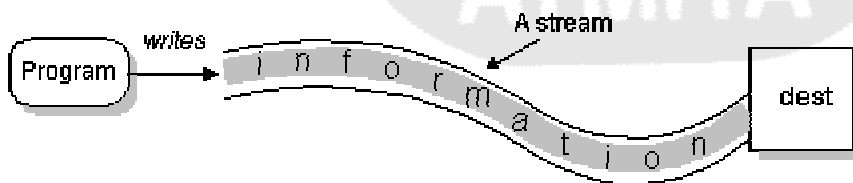
Often a program needs to bring in information from an external source or to send out information to an external destination. The information can be anywhere: in a file, on disk, somewhere on the network, in memory, or in another program. Also, the information can be of any type: objects, characters, images, or sounds. This chapter covers the Java™ platform classes that your programs can use to read and to write data.

### Basics

To bring in information, a program opens a *stream* on an information source (a file, memory, a socket) and reads the information sequentially, as shown here:



Similarly, a program can send information to an external destination by opening a stream to a destination and writing the information out sequentially, like this:



No matter where the data is coming from or going to and no matter what its type, the algorithms for sequentially reading and writing data are basically the same:

Reading	Writing
open a stream while more information	open a stream while more information

read information close the stream	write information close the stream
--------------------------------------	---------------------------------------

The java.io package contains a collection of stream classes that support these algorithms for reading and writing. To use these classes, a program needs to import the java.io package. The stream classes are divided into two class hierarchies, based on the data type (either characters or bytes) on which they operate.

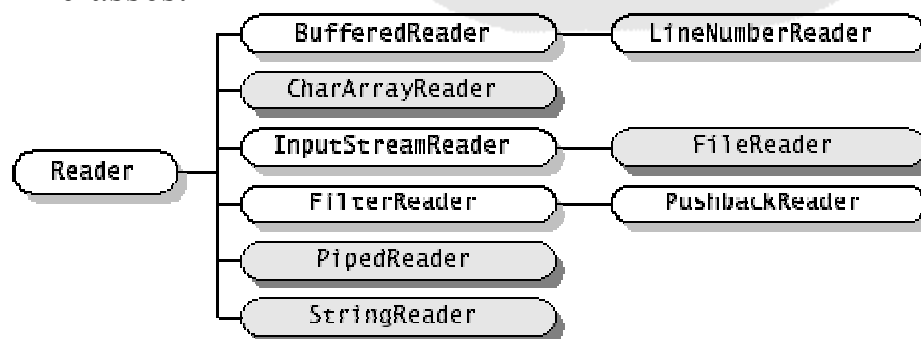
## Character Streams

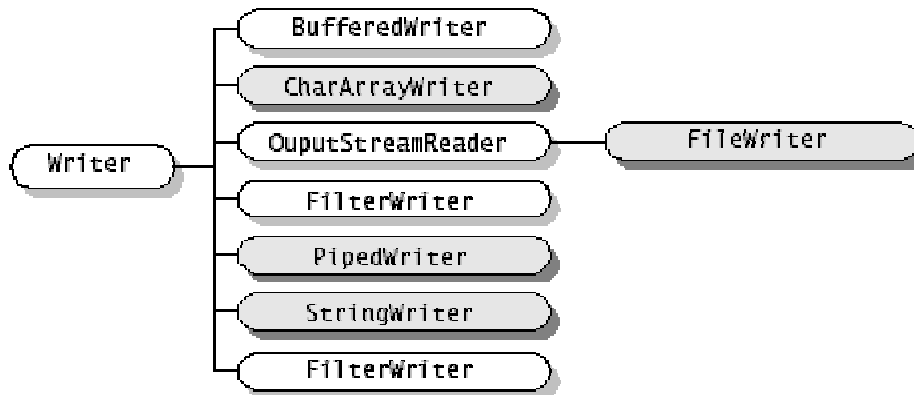
**Reader** and **Writer** are the abstract superclasses for character streams in java.io. **Reader** provides the API and implementation for readers-streams that read 16-bit characters and **Writer** provides the API and implementation for writers-streams that write 16-bit characters.

Subclasses of **Reader** and **Writer** implement specialized streams and are divided into two categories:

- those that read from or write to data sinks (shown in gray in the following figures)
- those that perform some sort of processing (shown in white).

The figure shows the class hierarchies for the **Reader** and **Writer** classes.





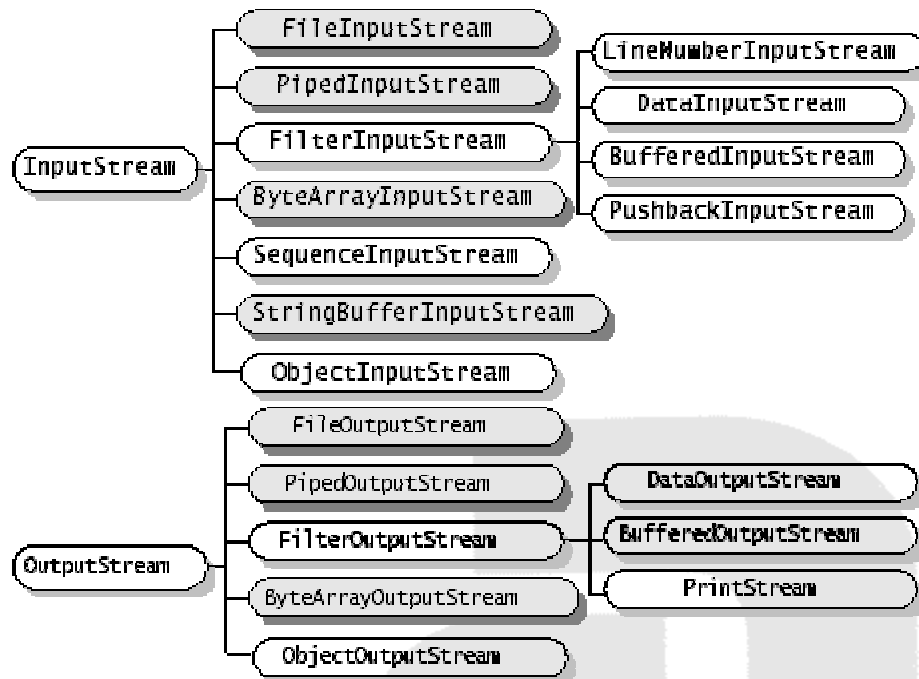
Most programs should use readers and writers to read and write textual information. The reason is that they can handle any character in the Unicode character set, whereas the byte streams are limited to 8-bit bytes.

### Byte Streams

To read and write 8-bit bytes, programs should use the byte streams, descendants of `InputStream` and `OutputStream`. `InputStream` and `OutputStream` provide the API and implementation for *input streams* (streams that read 8-bit bytes) and *output streams* (streams that write 8-bit bytes). These streams are typically used to read and write binary data such as images and sounds. Two of the byte stream classes, `ObjectInputStream` and `ObjectOutputStream`, are used for object serialization.

As with `Reader` and `Writer`, subclasses of `InputStream` and `OutputStream` provide specialized I/O that falls into two categories, as shown in the following class hierarchy figure:

- data sink streams (shaded)
- processing streams (unshaded).



## Understanding the I/O Superclasses

Reader and InputStream define similar APIs but for different data types. For example, Reader contains these methods for reading characters and arrays of characters:

- `int read()`
- `int read(char cbuf[])`
- `int read(char cbuf[], int offset, int length)`

InputStream defines the same methods but for reading bytes and arrays of bytes:

- `int read()`
- `int read(byte cbuf[])`
- `int read(byte cbuf[], int offset, int length)`

Also, both Reader and InputStream provide methods for marking a location in the stream, skipping input, and resetting the current position.

Writer and OutputStream are similarly parallel. Writer defines these methods for writing characters and arrays of characters:

- int write(int c)
- int write(char cbuf[])
- int write(char cbuf[], int offset, int length)

And OutputStream defines the same methods but for bytes:

- int write(int c)
- int write(byte cbuf[])
- int write(byte cbuf[], int offset, int length)

All of the streams-readers, writers, input streams, and output streams-are automatically opened when created. You can close any stream explicitly by calling its **close** method. Or the garbage collector can implicitly close it, which occurs when the object is no longer referenced.

### Using the Streams

The following table lists java.io's streams and describes what they do.

Note that many times, java.io contains character streams and byte streams that perform the same type of I/O but for different data types.

<u>I/O Streams</u>		
Type of I/O	Streams	Description
Memory	CharArrayReader	Use these streams to read from and write to memory. You create these streams on an existing array and then use the read and write methods to read from or write to the array.
	CharArrayWriter	
	ByteArrayInputStream	
	ByteArrayOutputStream	

	<p>StringReader</p> <p>StringWriter</p> <p>StringBufferInputStream</p>	<p>Use <b>StringReader</b> to read characters from a <b>String</b> in memory. Use <b>StringWriter</b> to write to a <b>String</b>. <b>StringWriter</b> collects the characters written to it in a <b>StringBuffer</b>, which can then be converted to a <b>String</b>.</p> <p><b>StringBufferInputStream</b> is similar to <b>StringReader</b>, except that it reads bytes from a <b>StringBuffer</b>.</p>
Pipe	<p>PipedReader</p> <p>PipedWriter</p> <p>PipedInputStream</p> <p>PipedOutputStream</p>	<p>Implement the input and output components of a pipe. Pipes are used to channel the output from one thread into the input of another.</p>
File	<p>FileReader</p> <p>FileWriter</p> <p>FileInputStream</p> <p>FileOutputStream</p>	<p>Collectively called file streams, these streams are used to read from or write to a file on the native file system.</p>
Concatenation	<p><i>N/A</i></p> <p>SequenceInputStream</p>	<p>Concatenates multiple input streams into one</p>

		input stream.
	<i>N/A</i>	
Object Serialization	ObjectInputStream ObjectOutputStream	Used to serialize objects.
	<i>N/A</i>	
Data Conversion	DataInputStream DataOutputStream	Read or write primitive data types in a machine-independent format.
Counting	LineNumberReader LineNumberInputStream	Keeps track of line numbers while reading.
Peeking Ahead	PushbackReader PushbackInputStream	These input streams each have a pushback buffer.
Printing	PrintWriter PrintStream	Contain convenient printing methods. These are the easiest streams to write to, so you will often see other writable streams wrapped in one of these.
Buffering	BufferedReader BufferedWriter BufferedInputStream BufferedOutputStream	Buffer data while reading or writing, thereby reducing the number of accesses required on the original data source. Buffered streams are typically more efficient than similar nonbuffered streams and are often used with other streams.

Filtering	FilterReader FilterWriter FilterInputStream FilterOutputStream	These abstract classes define the interface for filter streams, which filter data, as it's being read or written.  A reader and writer pair that forms the bridge between byte streams and character streams.  An InputStreamReader reads bytes from an InputStream and converts them to characters.  An OutputStreamWriter converts characters to bytes and then writes those bytes to an OutputStream.
Converting between Bytes and Characters	InputStreamReader OutputStreamWriter	

## Understanding the Implementation of various IO classes

### FileInputStream, FileOutputStream, FileReader, FileWriter

File streams are perhaps the easiest streams to understand. The file streams-- FileReader, FileWriter, FileInputStream, and



**FileOutputStream**--each read or write from a file on the native file system.

A **FileInputStream** obtains input bytes from a file in a file system. What files are available depends on the host environment.

A **FileOutputStream** is an output stream for writing data to a **File**. Whether or not a file is available or may be created depends upon the underlying platform. Some platforms, in particular, allow a file to be opened for writing by only one **FileOutputStream** at a time

**FileReader** is a convenience class for reading character files. The constructors of this class assume that the default character encoding and the default byte-buffer size are appropriate.

**FileWriter** is a convenience class for writing character files. The constructors of this class assume that the default character encoding and the default byte-buffer size are acceptable. To specify these values yourself, construct an **OutputStreamWriter** on a **FileOutputStream**.

```
import java.io.*;
public class Copy {
    public static void main(String[] args) throws IOException {
        File inputFile = new File("in.txt");
        File outputFile = new File("out.txt");

        FileReader in = new FileReader(inputFile);
        FileWriter out = new FileWriter(outputFile);
        int c;

        while ((c = in.read()) != -1)
            out.write(c);

        in.close();
    }
}
```

```
        out.close();
    }
}
/*streams....*/
import java.io.*;

public class CopyBytes {
    public static void main(String[] args) throws IOException {
        File inputFile = new File("farrago.txt");
        File outputFile = new File("outagain.txt");

        FileInputStream in = new FileInputStream(inputFile);
        FileOutputStream out = new FileOutputStream(outputFile);
        int c;

        while ((c = in.read()) != -1)
            out.write(c);

        in.close();
        out.close();
    }
}
```

This program is very simple. It opens a **FileReader** on **in.txt** and opens a **FileWriter** on **out.txt**. The program reads characters from the reader as long as there's more input in the input file and writes those characters to the writer. When the input runs out, the program closes both the reader and the writer.

## ByteArrayInputStream, ByteArrayOutputStream

A `ByteArrayInputStream` contains an internal buffer that contains bytes that may be read from the stream. An internal counter keeps track of the next byte to be supplied by the `read` method.

A `ByteArrayOutputStream` class implements an output stream in which the data is written into a byte array. The buffer automatically grows as data is written to it. The data can be retrieved using `toByteArray()` and `toString()`.

```
import java.io.*

public class ByteArrayIOApp {
    public static void main(String args[])
    throws IOException {
        ByteArrayOutputStream outputStream = new
        ByteArrayOutputStream();
        String s = "This is a test.";
        for(int i=0;i<s.length();++i)
            outputStream.write(s.charAt(i));
        System.out.println("outstream:
        "+outputStream);
        System.out.println("size:
        "+outputStream.size());
        ByteArrayInputStream inputStream;
        inputStream = new
        ByteArrayInputStream(outputStream.toByteArray());
        int inBytes = inputStream.available();
        System.out.println("inputStream has
        "+inBytes+" available bytes");
        byte inBuf[] = new byte[inBytes];
        int bytesRead =
```

```

inStream.read(inBuf, 0, inBytes);
    System.out.println(bytesRead+" bytes
were read");
    System.out.println("They are: "+new
String(inBuf, 0));
    }
}

```

## **InputStreamReader, OutputStreamWriter**

An `InputStreamReader` is a bridge from byte streams to character streams: It reads bytes and translates them into characters according to a specified character encoding. The encoding that it uses may be specified by name, or the platform's default encoding may be accepted.

An `OutputStreamWriter` is a bridge from character streams to byte streams: Characters written to it are translated into bytes according to a specified character encoding. The encoding that it uses may be specified by name, or the platform's default encoding may be accepted.

```

import java.io.*;
class WriteStuff {
    public static void main (String args[]) {
        // Copy the string into a byte array
        String s = new String("Dance, spider!\n");
        char[] buf = new char[64];
        s.getChars(0, s.length(), buf, 0);
        // Output the byte array (buffered)
        Writer out = new BufferedWriter(new
OutputStreamWriter(System.out));
        try {
            out.write(buf, 0, 64);
            out.flush();
        }
    }
}

```

```

        catch (Exception e) {
            System.out.println("Error: " + e.toString());
        }
    }
}

import java.io.*;
class ReadKeys4 {
    public static void main (String args[]) {
        Reader in = new BufferedReader(new
InputStreamReader(System.in));
        char buf[] = new char[10];
        try {
            in.read(buf, 0, 10);
        }
        catch (Exception e) {
            System.out.println("Error: " + e.toString());
        }
        String s = new String(buf);
        System.out.println(s);
    }
}

```

### **BufferedInputStream, BufferedOutputStream, BufferedReader, BufferedWriter**

A **BufferedInputStream** adds functionality to another input stream—namely, the ability to buffer the input and to support the **mark** and **reset** methods. When the **BufferedInputStream** is created, an internal buffer array is created. As bytes from the stream are read or skipped, the internal buffer is refilled as necessary from the contained input stream, many bytes at a time. The **mark** operation remembers a point in the input stream and the **reset** operation causes all the bytes read since the

most recent mark operation to be reread before new bytes are taken from the contained input stream.

A `BufferedOutputStream` class implements a buffered output stream. By setting up such an output stream, an application can write bytes to the underlying output stream without necessarily causing a call to the underlying system for each byte written. The data is written into an internal buffer, and then written to the underlying stream if the buffer reaches its capacity, the buffer output stream is closed, or the buffer output stream is explicitly flushed.

A `BufferedReader` reads text from a character-input stream, buffering characters so as to provide for the efficient reading of characters, arrays, and lines. The buffer size may be specified, or the default size may be used. The default is large enough for most purposes

A `BufferedWriter` writes text to a character-output stream, buffering characters so as to provide for the efficient writing of single characters, arrays, and strings. The buffer size may be specified, or the default size may be accepted. The default is large enough for most purposes.

```
import java.io.*;

public class BufferedIOApp {
    public static void main(String args[]) throws IOException {
        SequenceInputStream f3;
        FileInputStream f1 = new
FileInputStream("ByteArrayIOApp.java");
        FileInputStream f2 = new FileInputStream("FileIOApp.java");
        f3 = new SequenceInputStream(f1,f2);
        BufferedInputStream inStream = new BufferedInputStream(f3);
```

```
BufferedOutputStream bufStream = new
BufferedOutputStream(System.out);
PrintStream outputStream = new PrintStream(bufStream);
// inStream.skip(500);
boolean eof = false;
int byteCount = 0;
while (!eof) {
    int c = inStream.read();
    if(c == -1) eof = true;
    else{
        outputStream.print((char) c);
        ++byteCount;
    }
}
outputStream.println(byteCount+" bytes were read");
inStream.close();
outputStream.close();
f1.close();
f2.close();
}
}
```

## **DataInputStream, DataOutputStream**

A `DataInputStream` lets an application read primitive Java data types from an underlying input stream in a machine-independent way. An application uses a data output stream to write data that can later be read by a data input stream.

A `DataOutputStream` lets an application write primitive Java data types to an output stream in a portable way. An application can then use a data input stream to read the data back in.

```
import java.io.*;
```

```
class DataInputStreamDemo {  
  
    public static void main(String args[]) {  
  
        try {  
  
            // Create a file input stream  
            FileInputStream fis =  
                new FileInputStream(args[0]);  
  
            // Create a data input stream  
            DataInputStream dis =  
                new DataInputStream(fis);  
  
            // Read and display data  
            System.out.println(dis.readBoolean());  
            System.out.println(dis.readByte());  
            System.out.println(dis.readChar());  
            System.out.println(dis.readDouble());  
            System.out.println(dis.readFloat());  
            System.out.println(dis.readInt());  
            System.out.println(dis.readLong());  
            System.out.println(dis.readShort());  
  
            // Close file input stream  
            fis.close();  
        }  
        catch(Exception e) {  
            System.out.println("Exception: " + e);  
        }  
    }  
}  
  
import java.io.*;
```



```
class DataOutputStreamDemo {  
  
    public static void main(String args[]) {  
  
        try {  
  
            // Create a file output stream  
            FileOutputStream fos =  
                new FileOutputStream(args[0]);  
  
            // Create a data output stream  
            DataOutputStream dos =  
                new DataOutputStream(fos);  
  
            // Write various types of data  
            dos.writeBoolean(false);  
            dos.writeByte(Byte.MAX_VALUE);  
            dos.writeChar('A');  
            dos.writeDouble(Double.MAX_VALUE);  
            dos.writeFloat(Float.MAX_VALUE);  
            dos.writeInt(Integer.MAX_VALUE);  
            dos.writeLong(Long.MAX_VALUE);  
            dos.writeShort(Short.MAX_VALUE);  
  
            // Close file output stream  
            fos.close();  
        }  
        catch(Exception e) {  
            System.out.println("Exception: " + e);  
        }  
    }  
}
```

## SequenceInputStream

A SequenceInputStream represents the logical concatenation of other input streams. It starts out with an ordered collection of input streams and reads from the first one until end of file is reached, whereupon it reads from the second one, and so on, until end of file is reached on the last of the contained input streams.

//Listing 13.3. The source code of the SequenceIOApp program.

```
import java.io.*;

public class SequenceIOApp {
    public static void main(String args[]) throws IOException {
        SequenceInputStream inStream;
        FileInputStream f1 = new
FileInputStream("ByteArrayIOApp.java");
        FileInputStream f2 = new FileInputStream("FileIOApp.java");
        inStream = new SequenceInputStream(f1,f2);
        boolean eof = false;
        int byteCount = 0;
        while (!eof) {
            int c = inStream.read();
            if(c == -1) eof = true;
            else{
                System.out.print((char) c);
                ++byteCount;
            }
        }
        System.out.println(byteCount+" bytes were read");
        inStream.close();
        f1.close();
        f2.close();
    }
}
```

## CharArrayReader, CharArrayWriter

CharArrayReader class implements a character buffer that can be used as a character-input stream.

CharArrayWriter class implements a character buffer that can be used as an Writer. The buffer automatically grows when data is written to the stream. The data can be retrieved using toCharArray() and toString().

```
// Demonstrate CharArrayWriter.
import java.io.*;

class CharArrayWriterDemo {
    public static void main(String args[]) throws IOException {
        CharArrayWriter f = new CharArrayWriter();
        String s = "This should end up in the array";
        char buf[] = new char[s.length()];

        s.getChars(0, s.length(), buf, 0);
        f.write(buf);
        System.out.println("Buffer as a string");
        System.out.println(f.toString());
        System.out.println("Into array");

        char c[] = f.toCharArray();
        for (int i=0; i<c.length; i++) {
            System.out.print(c[i]);
        }

        System.out.println("\nTo a FileWriter()");
        FileWriter f2 = new FileWriter("test.txt");
        f.writeTo(f2);
        f2.close();
        System.out.println("Doing a reset");
        f.reset();
    }
}
```

```
    for (int i=0; i<3; i++)
        f.write('X');
    System.out.println(f.toString());
}
}

// Demonstrate CharArrayReader.
import java.io.*;

public class CharArrayReaderDemo {
    public static void main(String args[]) throws IOException {
        String tmp = "abcdefghijklmnopqrstuvwxyz";
        int length = tmp.length();
        char c[] = new char[length];

        tmp.getChars(0, length, c, 0);
        CharArrayReader input1 = new CharArrayReader(c);
        CharArrayReader input2 = new CharArrayReader(c, 0, 5);

        int i;
        System.out.println("input1 is:");
        while((i = input1.read()) != -1) {
            System.out.print((char)i);
        }
        System.out.println();

        System.out.println("input2 is:");
        while((i = input2.read()) != -1) {
            System.out.print((char)i);
        }
        System.out.println();
    }
}
```

## File, RandomAccessFile

File is an abstract representation of file and directory pathnames. User interfaces and operating systems use system-dependent *pathname strings* to name files and directories. This class presents an abstract, system-independent view of hierarchical pathnames.

RandomAccessFile class's instances support both reading and writing to a random access file. A random access file behaves like a large array of bytes stored in the file system. There is a kind of cursor, or index into the implied array, called the *file pointer*; input operations read bytes starting at the file pointer and advance the file pointer past the bytes read. The random access file is created in read/write mode.

```
import java.io.*;
class FileInfo {
    public static void main (String args[]) {
        System.out.println("Enter file name: ");
        char c;
        StringBuffer buf = new StringBuffer();
        try {
            Reader in = new InputStreamReader(System.in);
            while ((c = (char)in.read()) != '\n')
                buf.append(c);
        }
        catch (Exception e) {
            System.out.println("Error: " + e.toString());
        }
        File file = new File(buf.toString());
        System.out.println("sop :: "+buf.toString());

        System.out.println("File Name : " + file.getName());
        System.out.println("  Path : " + file.getPath());
        System.out.println("Abs. Path : " + file.getAbsolutePath());
        System.out.println("Writable : " + file.canWrite());
    }
}
```

```
        System.out.println("Readable   : " + file.canRead());
        System.out.println("Length     : " + (file.length() / 1024) + "KB");
    }

}
import java.io.*;

public class RandomIOApp {
    public static void main(String args[]) throws IOException {
        RandomAccessFile file = new RandomAccessFile("test.txt","rw");
        file.writeBoolean(true);
        file.writeInt(123456);
        file.writeChar('j');
        file.writeDouble(1234.56);
        file.seek(1);
        System.out.println(file.readInt());
        System.out.println(file.readChar());
        System.out.println(file.readDouble());
        file.seek(0);
        System.out.println(file.readBoolean());
        file.close();
    }
}
```

## **PrintWriter**

```
import java.io.*;
class PrintWriterDemo {
    public static void main(String args[]) {

        try {

            // Create a print writer
            PrintWriter pw = new PrintWriter(System.out);
```

```
// Experiment with some methods
pw.println(true);
pw.println('A');
pw.println(500);
pw.println(40000L);
pw.println(45.67f);
pw.println(45.67);
pw.println("Hello");
pw.println(new Integer("99"));

// Close print writer
pw.close();
}
catch(Exception e) {
    System.out.println("Exception: " + e);
}
}
}
```

## **PushbackInputStream, PushbackReader**

A `PushbackInputStream` adds functionality to another input stream, namely the ability to "push back" or "unread" one byte. This is useful in situations where it is convenient for a fragment of code to read an indefinite number of data bytes that are delimited by a particular byte value; after reading the terminating byte, the code fragment can "unread" it, so that the next read operation on the input stream will reread the byte that was pushed back. For example, bytes representing the characters constituting an identifier might be terminated by a byte representing an operator character; a method whose job is to read just an identifier can read until it sees the operator and then push the operator back to be re-read.

PushbackReader is a character-stream reader that allows characters to be pushed back into the stream.

## **ObjectInputStream, ObjectOutputStream**

An ObjectInputStream deserializes primitive data and objects previously written using an ObjectOutputStream. ObjectOutputStream and ObjectInputStream can provide an application with persistent storage for graphs of objects when used with a FileOutputStream and FileInputStream respectively. ObjectInputStream is used to recover those objects previously serialized. Other uses include passing objects between hosts using a socket stream or for marshaling and unmarshaling arguments and parameters in a remote communication system.

An ObjectOutputStream writes primitive data types and graphs of Java objects to an OutputStream (called Serialization). The objects can be read (reconstituted) using an ObjectInputStream. Persistent storage of objects can be accomplished by using a file for the stream. If the stream is a network socket stream, the objects can be reconstituted on another host or in another process.

```
import java.io.*;

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

        // Object serialization
        try {
            MyClass object1 = new MyClass("Hello", -7, 2.7e10);
            System.out.println("object1: " + object1);
        }
    }
}
```



```
        FileOutputStream fos = new FileOutputStream("serial");
        ObjectOutputStream oos = new ObjectOutputStream(fos);
        oos.writeObject(object1);
        oos.flush();
        oos.close();
    }
    catch(Exception e) {
        System.out.println("Exception during serialization: " + e);
        System.exit(0);
    }

// Object deserialization
try {
    MyClass object2;
    FileInputStream fis = new FileInputStream("serial");
    ObjectInputStream ois = new ObjectInputStream(fis);
    object2 = (MyClass)ois.readObject();
    ois.close();
    System.out.println("object2: " + object2);
}
catch(Exception e) {
    System.out.println("Exception during deserialization: " + e);
    System.exit(0);
}
}
}
```

```
class MyClass implements Serializable {
    String s;
    int i;
    double d;
    public MyClass(String s, int i, double d) {
        this.s = s;
        this.i = i;
        this.d = d;
    }
}
```

```
}  
public String toString() {  
    return "s=" + s + "; i=" + i + "; d=" + d;  
}
```

## **AWT Fundamentals**

### **Graphical User Interfaces**

The Java™ programming language provides a class library called the Abstract Window Toolkit (AWT) that contains a number of common graphical widgets. You can add these widgets to your display area and position them with a layout manager.

#### **AWT Basics**

All graphical user interface objects derived from a common superclass, Component. To create a Graphical User Interface (GUI), you add components to a Container object. Because a Container is also a Component, containers may be nested arbitrarily. Most often, you will use a Panel when creating nested GUIs.

Each AWT component uses native code to display itself on your screen. When you run a Java application under Microsoft Windows, buttons are really Microsoft Windows buttons. When you run the same application on a Macintosh, buttons are really Macintosh buttons. When you run on a UNIX machine that uses Motif, buttons are really Motif buttons.

#### **Applications versus Applets**

Recall that an Applet is a Java program that runs in a web page, while an application is one that runs from the command line. An Applet is a Panel that is automatically inserted into a web page. The browser displaying the web page instantiates and adds the

Applet to the proper part of the web page. The browser tells the Applet when to create its GUI (by calling the `init()` method of Applet) and when to `start()` and `stop()` any special processing. Applications run from a command prompt. When you execute an application from the command prompt, the interpreter starts by calling the application's `main()` method.

## Basic GUI Logic

There are three steps you take to create any GUI application or applet:

1. Compose your GUI by adding components to Container objects
2. Setup event handlers to respond to user interaction with the GUI
3. Display the GUI (automatically done for applets, you must explicitly do this for applications)

When you display an AWT GUI, the interpreter starts a new thread to watch for user interaction with the GUI. This new thread sits and waits until a user presses a key, clicks or moves the mouse, or any other system-level event that affects the GUI. When it receives such an event, it calls one of the event handlers you set up for the GUI. Note that the event handler code is executed within the thread that watches the GUI.

Because this extra thread exists, your main method can simply end after it displays the GUI. This makes GUI code very simple to write in AWT. Compose the GUI, setup event handlers, then display.

## A Simple Example

The following simple example shows some GUI code. This example creates an Applet.

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

/*<applet code="App1" width=200 height=200></applet>*/

class ABC implements ActionListener
{
    public void actionPerformed(ActionEvent ae)
    {
        Frame f=new Frame("Dreams Unlimited....");
        f.setLocation(300,400);
        f.setSize(200,100);
        f.setVisible(true);
    }
}

public class App1 extends Applet
{
    Button b;

    public void start()
    {
        b = new Button("OK");
        b.addActionListener(new ABC());

        add(b);
    }
}
```

When actionPerformed() is called, it generates a Frame

To try this applet, create a simple HTML page as follows.

```
<html>  
  <applet code=App1.class width=100 height=100>  
  </applet>  
</html>
```

Then test the HTML page by running appletviewer or by loading the HTML file in a browser that supports the Java Runtime Environment (JRE). Note that in this case, the browser must support at least version 1.1 of the JRE, as the example uses the event handling capabilities introduced with that release.

## **AWT Components**

All AWT components extend class Component. Think of Component as the "root of all evil" for AWT. Having this single class is rather useful, as the library designers can put a lot of common code into it.

Next, examine each of the AWT components below. Most, but not all, directly extend Component. You've used most of the components should be familiar to you.

### **Buttons**

A Button has a single line label and may be "pushed" with a mouse click.

```
import java.awt.*;  
import java.applet.Applet;  
  
public class ButtonTest extends Applet {  
  public void init() {  
    Button button = new Button("OK");  
    add(button);  
  }  
}
```

```
}  
}
```

Note that in the above example there is no event handling added; pressing the button will not do anything.

The AWT button has no direct support for images as labels.

## Canvas

A Canvas is a graphical component representing a region where you can draw things such as rectangles, circles, and text strings. The name comes from a painter's canvas. You subclass Canvas to override its default `paint()` method to define your own components.

You can subclass Canvas to provide a custom graphic in an applet.

```
import java.awt.Canvas;  
import java.awt.Graphics;  
  
class DrawingRegion extends Canvas {  
    public DrawingRegion() {  
        setSize(100, 50);  
    }  
    public void paint(Graphics g) {  
        g.drawRect(0, 0, 99, 49); // draw border  
        g.drawString("A Canvas", 20,20);  
    }  
}
```

Then you use it like any other component, adding it to a parent container, for example in an Applet subclass.

```
import java.applet.Applet;
```

```
public class CanvasPaintTest extends Applet {
    public void init() {
        DrawingRegion region = new DrawingRegion();
        add(region);
    }
}
```

The Canvas class is frequently extended to create new component types, for example image buttons. However, starting with the JRE 1.1, you can now directly subclass Component directly to create lightweight, transparent widgets.

### Checkbox

A Checkbox is a label with a small pushbutton. The state of a Checkbox is either true (button is checked) or false (button not checked). The default initial state is false. Clicking a Checkbox toggles its state. For example:

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

public class CheckboxSimpleTest extends Applet {
    public void init() {
        Checkbox m = new Checkbox("Allow Mixed Case");
        add(m);
    }
}
```

To set a Checkbox initially true use an alternate constructor:

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

public class CheckboxSimpleTest2 extends Applet {
    public void init() {
```

```

    Checkbox m = new Checkbox("Label", true);
    add(m);
}
}

```

## CheckboxGroup

A CheckboxGroup is used to control the behavior of a group of Checkbox objects (each of which has a true or false state). Exactly one of the Checkbox objects is allowed to be true at one time. Checkbox objects controlled with a CheckboxGroup are usually referred to as "radio buttons". The following example illustrates the basic idea behind radio buttons.

```

import java.awt.*;
import java.applet.Applet;

public class CheckboxGroupTest extends Applet {
    public void init() {
        // create button controller
        CheckboxGroup cbg = new CheckboxGroup();

        Checkbox cb1 =
            new Checkbox("Show lowercase only", cbg, true);
        Checkbox cb2 =
            new Checkbox("Show uppercase only", cbg, false);

        add(cb1);
        add(cb2);
    }
}

```



## Choice

Choice objects are drop-down lists. The visible label of the Choice object is the currently selected entry of the Choice.

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

public class ChoiceSimpleTest extends Applet {
    public void init() {
        Choice rgb = new Choice();

        rgb.add("Red");
        rgb.add("Green");
        rgb.add("Blue");

        add(rgb);
    }
}
```

The first item added is the initial selection.

## Label

A Label is a displayed Label object. It is usually used to help indicate what other parts of the GUI do, such as the purpose of a neighboring text field.

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

public class LabelTest extends Applet {
    public void init() {
        add(new Label("A label"));
    }
}
```

```
    // right justify next label
    add(new Label("Another label", Label.RIGHT));
}
}
```

Like the Button component, a Label is restricted to a single line of text.

## List

A List is a scrolling list box that allows you to select one or more items.

Multiple selections may be used by passing true as the second argument to the constructor.

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

public class ListSimpleTest extends Applet {
    public void init() {
        List list = new List(5, false);
        list.add("Seattle");
        list.add("Washington");
        list.add("New York");
        list.add("Chicago");
        list.add("Miami");
        list.add("San Jose");
        list.add("Denver");

        add(list);
    }
}
```

The constructor may contain a preferred number of lines to display. The current LayoutManager may choose to respect or ignore this request.

## Scrollbar

A Scrollbar is a "slider" widget with characteristics specified by integer values that are set during Scrollbar construction. Both horizontal and vertical sliders are available.

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

// A simple example that makes a Scrollbar appear
public class ScrollbarSimpleTest extends Applet {
    public void init() {
        Scrollbar sb =
            new Scrollbar(Scrollbar.HORIZONTAL,
                0, // initial value is 0
                5, // width of slider
                -100, 105); // range -100 to 100
        add(sb);
    }
}
```

The maximum value of the Scrollbar is determined by subtracting the Scrollbar width from the maximum setting (last parameter).

## TextField

A TextField is a scrollable text display object with one row of characters. The preferred width of the field may be specified during construction and an initial string may be specified.

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

public class TextFieldSimpleTest extends Applet {
```

```
public void init() {  
    TextField f1 =  
        new TextField("type something");  
    add(f1);  
}  
}
```

Tips:

- Call `setEditable(true)` to make the field read-only.
- The constructor has an optional width parameter.

This does not control the number of characters in the `TextField`, but is merely a suggestion of the preferred width on the screen. Note that layout managers may choose to respect or ignore this preferred width.

For password fields:

```
field.setEchoChar('?');
```

To clear/reset:

```
field.setEchoChar((char)0);
```

## **TextArea**

A `TextArea` is a multi-row text field that displays a single string of characters, where newline (`\n` or `\n\r` or `\r`, depending on platform) ends each row. The width and height of the field is set at construction, but the text can be scrolled up/down and left/right.

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

```
import java.applet.Applet;
```

```
public class TextAreaSimpleTest extends Applet {  
    TextArea disp;  
    public void init() {  
        disp = new TextArea("Code goes here", 10, 30);  
        add(disp);  
    }  
}
```

}

There is no way, for example, to put the cursor at beginning of row five, only to put the cursor at single dimension position 50. There is a four-argument constructor that accepts a fourth parameter of a scrollbar policy. The different settings are the class constants: `SCROLLBARS_BOTH`, `SCROLLBARS_HORIZONTAL_ONLY`, `SCROLLBARS_NONE`, and `SCROLLBARS_VERTICAL_ONLY`. When the horizontal (bottom) scrollbar is not present, the text will wrap.

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

public class TextAreaScroll extends Applet {
    String s =
        "This is a very long message " +
        "It should wrap when there is " +
        "no horizontal scrollbar.";
    public void init() {
        add(new TextArea (s, 4, 15,
            TextArea.SCROLLBARS_NONE));
        add(new TextArea (s, 4, 15,
            TextArea.SCROLLBARS_BOTH));
        add(new TextArea (s, 4, 15,
            TextArea.SCROLLBARS_HORIZONTAL_ONLY));
        add(new TextArea (s, 4, 15,
            TextArea.SCROLLBARS_VERTICAL_ONLY));
    }
}
```

## Common Component Methods

All AWT components share the 100-plus methods inherited from the Component class. Some of the most useful and commonly-used methods are listed below:

`getSize()` - Gets current size of component, as a Dimension.

- `Dimension d = someComponent.getSize();`
- `int height = d.height;`
- `int width = d.width;`

Note: With the Java 2 Platform, you can directly access the width and height using the `getWidth()` and `getHeight()` methods. This is more efficient, as the component doesn't need to create a new Dimension object. For example:

```
int height = someComponent.getHeight();
int width = someComponent.getWidth();
```

if you need a Dimension object, you should only use `getSize()`

`getLocation()` - Gets position of component, relative to containing component, as a Point.

```
Point p = someComponent.getLocation();
int x = p.x;
int y = p.y;
```

Note: With the Java 2 Platform, you can directly access the x and y parts of the location using `getX()` and `getY()`. This is more efficient, as the component doesn't have to create a new Point object. For example:

```
int x = someComponent.getX();
int y = someComponent.getY();
```

If you're using the Java 2 platform, you should only use `getLocation()` if you really need a `Point` object  
`getLocationOnScreen()` - Gets the position of the component relative to the upper-left corner of the computer screen, as a `Point`.

```
Point p = someComponent.getLocationOnScreen();
int x = p.x;
int y = p.y;
```

- `getBounds()` - Gets current bounding Rectangle of component.

```
Rectangle r = someComponent.getBounds();
int height = r.height;
int width = r.width;
int x = r.x;
int y = r.y;
```

This is like a combination of calling `getLocation()` and `getSize()`.  
Note: If you're using the Java 2 Platform and don't really need a `Rectangle` object, you should use `getX()`, `getY()`, `getWidth()`, and `getHeight()` instead.

- • `setEnabled(boolean)` - Toggles the state of the component.

If set to true, the component will react to user input and appear normal. If set to false, the component will ignore user interaction, and usually appear ghosted or grayed-out.

- • `setVisible(boolean)` - Toggles the visibility state of the component.

If set to true, the component will appear on the screen if it is contained in a visible container. If false, the component will not appear on the screen.

Note that if a component is marked as not visible, any layout manager that is responsible for that component will usually proceed with the layout algorithm as though the component were not in the parent container! This means that making a component invisible will not simply make it disappear while reserving its space in the GUI. Making the component invisible will cause the layout of its sibling components to readjust.

- • setBackground(Color)/setForeground(Color) - Changes component background/foreground colors.
- • setFont(Font) - Changes font of text within component.

## Containers

A Container is a Component, so may be nested. Class Panel is the most commonly-used Panel and can be extended to partition GUIs. Class Applet is a specialized Panel for running programs within a browser.

## Common Container Methods

Besides the 100-plus methods inherited from the Component class, all Container subclasses inherit the behavior of about 50 common methods of Container (most of which just override a method of Component). While the most common method of Container used add(), has already been briefly discussed, if you need to access the list of components within a container, you may



find the `getComponentCount()`, `getComponents()`, and `getComponent(int)` methods helpful.

## ScrollPane

The `ScrollPane` container was introduced with the 1.1 release of the Java Runtime Environment (JRE) to provide a new Container with automatic scrolling of any one large Component. That large object could be anything from an image that is too big for the display area to a bunch of spreadsheet cells. All the event handling mechanisms for scrolling are managed for you. Also, there is no `LayoutManager` for a `ScrollPane` since there is only a single object within it.

The following example demonstrates the scrolling of a large image. Since an `Image` object is not a Component, the image must be drawn by a component such as a `Canvas`.

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

```
class ImageCanvas extends Component {  
    private Image image;  
    public ImageCanvas(Image i) {  
        image = i;  
    }  
    public void paint(Graphics g) {  
        if (image != null)  
            g.drawImage(image, 0, 0, this);  
    }  
}
```

```
public class ScrollingImage extends Applet {  
    public void init() {  
        setLayout(new BorderLayout());  
        ScrollPane sp =  
            new ScrollPane(ScrollPane.SCROLLBARS_ALWAYS);
```

```
    Image im =
        getImage(getCodeBase(), "./images/kid.gif");
    sp.add(new ImageCanvas(im));
    add(sp, BorderLayout.CENTER);
}
}
```

## Event Handling

### Events

Beginning with the 1.1 version of the JRE, objects register as listeners for events. If there are no listeners when an event happens, nothing happens. If there are twenty listeners registered, each is given an opportunity to process the event, in an undefined order. With a Button, for example, activating the button notifies any registered ActionListener objects. Consider SimpleButtonEvent applet which creates a Button instance and registers itself as the listener for the button's action events:

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

public class SimpleButtonEvent extends Applet
    implements ActionListener {
    private Button b;

    public void init() {
        b = new Button("Press me");
        b.addActionListener(this);
        add(b);
    }

    public void actionPerformed(ActionEvent e) {
```

```
// If the target of the event was our Button
// In this example, the check is not
// truly necessary as we only listen to
// a single button
if ( e.getSource() == b ) {
    getGraphics().drawString("OUCH",20,20);
}
}
```

Notice that any class can implement ActionListener, including, in this case, the applet itself. All listeners are always notified.

So, here is how everything works:

Components generate subclasses of AWTEvent when something interesting happens.

Event sources permit any class to be a listener using the addXXXListener() method, where XXX is the event type you can listen for, for example addActionListener(). You can also remove listeners using the removeXXXListener() methods. If there is an add/removeXXXListener() pair, then the component is a source for the event when the appropriate action happens.

In order to be an event handler you have to implement the listener type, otherwise, you cannot be added, ActionListener being one such type.

Some listener types are special and require you to implement multiple methods. For instance, if you are interested in key events, and register a KeyListener, you have to implement three methods, one for key press, one for key release, and one for both, key typed. If you only care about key typed events, it doesn't make sense to have to stub out the other two methods. There are special classes out there called adapters that implement the listener interfaces and stub out all the methods. Then, you only need to subclass the adapter and override the necessary method(s).

## AWTEvent

Events subclass the AWTEvent class. And nearly every event-type has an associated Listener interface, PaintEvent and InputEvent do not. (With PaintEvent, you just override paint() and update(), for InputEvent, you listen for subclass events, since it is abstract.

### Low-level Events

Low-level events represent a low-level input or window operation, like a key press, mouse movement, or window opening. The following table displays the different low-level events, and the operations that generate each event (each operation corresponds to a method of the listener interface):

ComponentEvent	Hiding, moving, resizing, showing
ContainerEvent	Adding/removing component
FocusEvent	Getting/losing focus
KeyEvent	Pressing, releasing, or typing (both) a key
MouseEvent	Clicking, dragging, entering, exiting, moving, pressing, or releasing
WindowEvent	Iconifying, deiconifying, opening, closing, really closed, activating, deactivating

For instance, typing the letter 'A' on the keyboard generates three events, one for pressing, one for releasing, and one for typing.

Depending upon your interests, you can do something for any of the three events.

### Semantic Events

Semantic events represent interaction with a GUI component; for instance selecting a button, or changing the text of a text field. Which components generate which events is shown in the next section.

ActionEvent	Do the command
AdjustmentEvent	Value adjusted
ItemEvent	State changed
TextEvent	Text changed

### Event Sources

The following table represents the different event sources. Keep in mind the object hierarchy. For instance, when Component is an event source for something, so are all its subclasses:

### Low-Level Events

Component	ComponentListener KeyListener MouseMotionListener FocusListener
-----------	--

	MouseListener
Container	ContainerListener
Window	WindowListener

## Semantic Events

ActionListener	List TextField Button MenuItem
Checkbox Choice CheckboxMenuItem List	ItemListener
Scrollbar	AdjustmentListener
TextArea TextListener	TextField

Notice that although there is only one MouseEvent class, the listeners are spread across two interfaces. This is for performance issues. Since motion mouse events are generated more frequently, if you have no interest in them, you can ignore them more easily, without the performance hit.

## Event Listeners

Each listener interface is paired with one event type and contains a method for each type of event the event class embodies. For instance, the KeyListener contains three methods, one for each type of event that the KeyEvent has: keyPressed(), keyReleased(), and keyTyped().

**Summary of Listener interfaces and their methods**

<b>Interface Method(s)</b>	
ActionListener	actionPerformed(ActionEvent e)
AdjustmentListener	adjustmentValueChanged(AdjustmentEvent e)
ComponentListener	componentHidden(ComponentEvent e)
	componentMoved(ComponentEvent e)
	componentResized(ComponentEvent e)
	componentShown(ComponentEvent e)
ContainerListener	componentAdded(ContainerEvent e)
	componentRemoved(ContainerEvent e)
FocusListener	FocusGained(FocusEvent e)
	focusLost(FocusEvent e)
ItemListener	itemStateChanged(ItemEvent e)
KeyListener	keyPressed(KeyEvent e)
	keyReleased(KeyEvent e)
	keyTyped(KeyEvent e)
MouseListener	mouseClicked(MouseEvent e)
	mouseEntered(MouseEvent e)
	mouseExited(MouseEvent e)

	mousePressed(MouseEvent e)
	mouseReleased(MouseEvent e)
MouseMotionListener	mouseDragged(MouseEvent e)
	mouseMoved(MouseEvent e)
TextListener	textValueChanged(TextEvent e)
WindowListener	windowActivated(WindowEvent e)
	windowClosed(WindowEvent e)
	windowClosing(WindowEvent e)
	windowDeactivated(WindowEvent e)
	windowDeiconified(WindowEvent e)
	windowIconified(WindowEvent e)
	windowOpened(WindowEvent e)

### Event Adapters

Since the low-level event listeners have multiple methods to implement, there are event adapter classes to ease the pain. Instead of implementing the interface and stubbing out the methods you do not care about, you can subclass the appropriate adapter class and just override the one or two methods you are interested in. Since the semantic listeners only contain one method to implement, there is no need for adapter classes.



```

public class MyKeyAdapter extends KeyAdapter {
    public void keyTyped(KeyEvent e) {
        System.out.println("User typed: " +
            KeyEvent.getKeyText(e.getKeyCode()));
    }
}

```

### Button Pressing Example

The following code demonstrates the basic concept a little more beyond the earlier example. There are three buttons within a Frame, their displayed labels may be internationalized so you need to preserve their purpose within a command associated with the button. Based upon which button is pressed, a different action occurs.

```

import java.awt.*;
import java.awt.event.*;

public class Activator {
    public static void main(String[] args) {
        Button b;
        ActionListener al = new MyActionListener();
        Frame f = new Frame("Hello Java");
        f.add(b = new Button("Hola"), BorderLayout.NORTH);
        b.setActionCommand("Hello");
        b.addActionListener(al);
        f.add(b = new Button("Aloha"), BorderLayout.CENTER);
        b.addActionListener(al);
        f.add(b = new Button("Adios"), BorderLayout.SOUTH);
        b.setActionCommand("Quit");
        b.addActionListener(al);
        f.pack();
        f.show();
    }
}

```

```

class MyActionListener implements ActionListener {
    public void actionPerformed(ActionEvent e) {
        // Action Command is not necessarily label
        String s = e.getActionCommand();
        if (s.equals("Quit")) {
            System.exit(0);
        }
        else if (s.equals("Hello")) {
            System.out.println("Bon Jour");
        }
        else {
            System.out.println(s + " selected");
        }
    }
}

```

Since this is an application, you need to save the source (as Activator.java), compile, and run it.

### **Adapters Example**

The following code demonstrates using an adapter as an anonymous inner class to draw a rectangle within an applet. The mouse press signifies the top left corner to draw, with the mouse release the bottom right.

```

import java.awt.*;
import java.awt.event.*;

public class Draw extends java.applet.Applet {
    public void init() {
        addMouseListener(
            new MouseAdapter() {
                int savedX, savedY;
            }
        );
    }
}

```

```

        public void mousePressed(MouseEvent e) {
            savedX = e.getX();
            savedY = e.getY();
        }
        public void mouseReleased(MouseEvent e) {
            Graphics g = Draw.this.getGraphics();
            g.drawRect(savedX, savedY,
                e.getX()-savedX,
                e.getY()-savedY);
        }
    }
};
}
}

```

## Applications and Menus

### GUI-based Applications

To create a window for your application, define a subclass of `Frame` (a `Window` with a title, menubar, and border) and have the main method construct an instance of that class.

Applications respond to events in the same way as applets do. The following example, `BasicApplication`, responds to the native window toolkit quit, or closing, operation:

```

import java.awt.*;
import java.awt.event.*;

public class BasicApplication extends Frame {
    public BasicApplication() {
        super("BasicApplication Title");
        setSize(200, 200);
        // add a demo component to this frame
        add(new Label("Application Template...", Label.CENTER),

```

```

        BorderLayout.CENTER);
    addWindowListener(new WindowAdapter() {
        public void windowClosing(WindowEvent e) {
            setVisible(false); dispose();
            System.exit(0);
        }
    });
}

public static void main(String[] args) {
    BasicApplication app = new BasicApplication();
    app.setVisible(true);
}
}

```

Consider an application that displays the x,y location of the last mouse click and provides a button to reset the displayed x,y coordinates to 0,0:

```

import java.awt.*;
import java.awt.event.*;
public class CursorFrame extends Frame {
    TextField a, b;
    Button btn;
    public CursorFrame() {
        super("CursorFrame");
        setSize(400, 200);
        setLayout(new FlowLayout());
        add(new Label("Click the mouse..."));
        a = new TextField("0", 4);
        b = new TextField("0", 4);
        btn = new Button("RESET");
        add(a); add(b); add(btn);
        addMouseListener(new MouseAdapter() {

```

```

    public void mousePressed(MouseEvent e) {
        a.setText(String.valueOf(e.getX()));
        b.setText(String.valueOf(e.getY()));
    }
});
addWindowListener(new WindowAdapter() {
    public void windowClosing(WindowEvent e) {
        setVisible(false);
        dispose();
        System.exit(0);
    }
});
btn.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent e) {
        a.setText("0");
        b.setText("0");
    }
});
}
public static void main(String[] args) {
    CursorFrame app = new CursorFrame();
    app.setVisible(true);
}
}

```

This application provides anonymous classes to handle mouse events, application window closing events, and the action event for resetting the text fields that report mouse coordinates.

When you have a very common operation, such as handling application window closing events, it often makes sense to abstract out this behavior and handle it elsewhere. In this case, it's logical to do this by extending the existing Frame class, creating the specialization AppFrame:

```
import java.awt.*;
import java.awt.event.*;

public class AppFrame extends Frame
    implements WindowListener {
    public AppFrame(String title) {
        super(title);
        addWindowListener(this);
    }
    public void windowClosing(WindowEvent e) {
        setVisible(false);
        dispose();
        System.exit(0);
    }
    public void windowClosed(WindowEvent e) {}
    public void windowDeactivated(WindowEvent e) {}
    public void windowActivated(WindowEvent e) {}
    public void windowDeiconified(WindowEvent e) {}
    public void windowIconified(WindowEvent e) {}
    public void windowOpened(WindowEvent e) {}
}
```

AppFrame directly implements WindowListener, providing empty methods for all but one window event, namely, the window closing operation. With this definition, applications such as CursorFrame can extend AppFrame instead of Frame and avoid having to provide the anonymous class for window closing operations:

### **Applications: Dialog Boxes**

A Dialog is a window that requires input from the user. Components may be added to the Dialog like any other container. Like a Frame, a Dialog is initially invisible. You must call the method setVisible() to activate the dialog box.

```
import java.awt.*;
import java.awt.event.*;

public class DialogFrame extends AppFrame {
    Dialog d;

    public DialogFrame() {
        super("DialogFrame");
        setSize(200, 100);
        Button btn, dbtn;
        add(btn = new Button("Press for Dialog Box"),
            BorderLayout.SOUTH);
        d = new Dialog(this, "Dialog Box", false);
        d.setSize(150, 150);
        d.add(new Label("This is the dialog box."),
            BorderLayout.CENTER);
        d.add(dbtn = new Button("OK"),
            BorderLayout.SOUTH);
        btn.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent e) {
                d.setVisible(true);
            }
        });
        dbtn.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent e) {
                d.setVisible(false);
            }
        });
        d.addWindowListener(new WindowAdapter() {
            public void windowClosing(WindowEvent e) {
                d.setVisible(false);
            }
        });
    }
}
```

```

public static void main(String[] args) {
    DialogFrame app = new DialogFrame();
    app.setVisible(true);
}
}

```

Again, you can define anonymous classes on the fly for:

1. Activating the dialog window from the main application's command button.
2. Deactivating the dialog window from the dialog's command button.
3. Deactivating the dialog window in response to a native window system's closing operation.

Although the anonymous class functionality is quite elegant, it is inconvenient to have to repeatedly include the window-closing functionality for every dialog instance that your applications instantiate by coding and registering the anonymous window adapter class. As with AppFrame, you can define a specialization of Dialog that adds this functionality and thereafter simply use the enhanced class. For example, WMDialog provides this functionality:

```

import java.awt.*;
import java.awt.event.*;
public class WMDialog extends Dialog
    implements WindowListener {
    public WMDialog(Frame ref, String title, boolean modal) {
        super(ref, title, modal);
        addWindowListener(this);
    }
    public void windowClosing(WindowEvent e) {
        setVisible(false);
    }
    public void windowClosed(WindowEvent e) {}
}

```



```

public void windowDeactivated(WindowEvent e) {}
public void windowActivated(WindowEvent e) {}
public void windowDeiconified(WindowEvent e) {}
public void windowIconified(WindowEvent e) {}
public void windowOpened(WindowEvent e) {}
}

```

### Applications: Menus

An application can have a MenuBar object containing Menu objects that are comprised of MenuItem objects. Each MenuItem can be a string, menu, checkbox, or separator (a line across the menu).

To add menus to any Frame or subclass of Frame:

1. Create a MenuBar
- 2.
3. MenuBar mb = new MenuBar();
4. Create a Menu
- 5.
6. Menu m = new Menu("File");
7. Create your MenuItem choices and add each to the Menu, in the order you want them to appear, from top to bottom.
- 8.
9. m.add(new MenuItem("Open"));
10. m.addSeparator(); // add a separator
11. m.add(new CheckboxMenuItem("Allow writing"));
12. // Create submenu
13. Menu sub = new Menu("Options...");
14. sub.add(new MenuItem("Option 1"));
15. m.add(sub); // add sub to File menu
16. Add each Menu to the MenuBar in the order you want them to appear, from left to right.
- 17.
18. mb.add(m); // add File menu to bar

19. Add the MenuBar to the Frame by calling the setMenuBar() method .

20.

21. setMenuBar(mb); // set menu bar of your Frame

The following program, MainWindow, creates an application window with a menu bar and several menus using the strategy outlined above:

```
import java.awt.*;  
import java.awt.event.*;
```

```
// Make a main window with two top-level menus: File and Help.  
// Help has a submenu and demonstrates a few interesting menu  
items.
```

```
public class MainWindow extends Frame {  
    public MainWindow() {  
        super("Menu System Test Window");  
        setSize(200, 200);  
  
        // make a top level File menu  
        FileMenu fileMenu = new FileMenu(this);  
  
        // make a top level Help menu  
        HelpMenu helpMenu = new HelpMenu(this);  
  
        // make a menu bar for this frame  
        // and add top level menus File and Menu  
        MenuBar mb = new MenuBar();  
        mb.add(fileMenu);  
        mb.add(helpMenu);  
        setMenuBar(mb);  
        addWindowListener(new WindowAdapter() {  
            public void windowClosing(WindowEvent e) {  
                exit();  
            }  
        });  
    }  
}
```

```

    }
  });
}

public void exit() {
    setVisible(false); // hide the Frame
    dispose(); // tell windowing system to free resources
    System.exit(0); // exit
}

public static void main(String args[]) {
    MainWindow w = new MainWindow();
    w.setVisible(true);
}
}

// Encapsulate the look and behavior of the File menu
class FileMenu extends Menu implements ActionListener {
    MainWindow mw; // who owns us?
    public FileMenu(MainWindow m) {
        super("File");
        mw = m;
        MenuItem mi;
        add(mi = new MenuItem("Open"));
        mi.addActionListener(this);
        add(mi = new MenuItem("Close"));
        mi.addActionListener(this);
        add(mi = new MenuItem("Exit"));
        mi.addActionListener(this);
    }
    // respond to the Exit menu choice
    public void actionPerformed(ActionEvent e) {
        String item = e.getActionCommand();
        if (item.equals("Exit"))
            mw.exit();
    }
}

```

```

    else
        System.out.println("Selected FileMenu " + item);
    }
}

// Encapsulate the look and behavior of the Help menu
class HelpMenu extends Menu implements ActionListener {
    MainWindow mw; // who owns us?
    public HelpMenu(MainWindow m) {
        super("Help");
        mw = m;
        MenuItem mi;
        add(mi = new MenuItem("Fundamentals"));
        mi.addActionListener(this);
        add(mi = new MenuItem("Advanced"));
        mi.addActionListener(this);
        addSeparator();
        add(mi = new CheckboxMenuItem("Have Read The Manual"));
        mi.addActionListener(this);
        add(mi = new CheckboxMenuItem("Have Not Read The
Manual"));
        mi.addActionListener(this);

        // make a Misc sub menu of Help menu
        Menu subMenu = new Menu("Misc");
        subMenu.add(mi = new MenuItem("Help!!!"));
        mi.addActionListener(this);
        subMenu.add(mi = new MenuItem("Why did that happen?"));
        mi.addActionListener(this);
        add(subMenu);
    }
    // respond to a few menu items
    public void actionPerformed(ActionEvent e) {
        String item = e.getActionCommand();
        if (item.equals("Fundamentals"))

```

```
        System.out.println("Fundamentals");
    else if (item.equals("Help!!!"))
        System.out.println("Help!!!");
    // etc...
}
}
```

## Menu Shortcuts

One nice feature of the MenuItem class is its ability to provide menu shortcuts or speed keys. For instance, in most applications that provide printing capabilities, pressing Ctrl-P initiates the printing process. When you create a MenuItem you can specify the shortcut associated with it. If the user happens to press the speed key, the action event is triggered for the menu item. If you want to create two menu items with speed keys, Ctrl-P for Print and Shift-Ctrl-P for Print Preview, the following code would do that:

```
file.add (mi = new MenuItem ("Print",
    new MenuShortcut('p')));
file.add (mi = new MenuItem ("Print Preview",
    new MenuShortcut('p', true)));
```

The example above uses Ctrl-P and Shift-Ctrl-P shortcuts on Windows/Motif. The use of Ctrl for the shortcut key is defined by the Toolkit method

getMenuShortcutKeyMask(). For the Macintosh, this would be the Command key. An optional boolean parameter to the constructor determines the need for the Shift key appropriate to the platform.

## Pop-up Menus

One restriction of the Menu class is that it can only be added to a Frame. If you want a menu in an Applet, you are out of luck (unless you use the Swing component set). While not necessarily a perfect solution, you can associate a pop-up menu with any Component, of which Applet is a subclass. A PopupMenu is similar to a Menu in that it holds MenuItem objects. However, instead of appearing at the top of a Frame, you pop the popup menu up over any component, usually when the user generates the appropriate mouse event.

The actual mouse interaction to generate the event is platform specific so there is the means to determine if a MouseEvent triggers the pop-up menu via the MouseEvent.isPopupTrigger() method. It is then your responsibility to position and display the PopupMenu.

The following program, PopupApplication, demonstrates this portable triggering of a pop-up menu, as well as activating a pop-up menu from a command button:

```
import java.awt.*;
import java.awt.event.*;
public class PopupApplication extends AppFrame {
    Button btn; TextField msg; PopupAppMenu m;
    public PopupApplication() {
        super("PopupApplication");
        setSize(200, 200);
        btn = new Button("Press for pop-up menu...");
        add(btn, BorderLayout.NORTH);
        msg = new TextField();
        msg.setEditable(false);
        add(msg, BorderLayout.SOUTH);
        m = new PopupAppMenu(this);
    }
}
```

```

add(m);
btn.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent e) {
        m.show(btn, 10, 10);
    }
});
addMouseListener(new MouseAdapter() {
    public void mousePressed(MouseEvent e) {
        if (e.isPopupTrigger())
            m.show(e.getComponent(), e.getX(), e.getY());
    }
    public void mouseReleased(MouseEvent e) {
        if (e.isPopupTrigger())
            m.show(e.getComponent(), e.getX(), e.getY());
    }
});
}
public static void main(String[] args) {
    PopupApplication app = new PopupApplication();
    app.setVisible(true);
}
}

```

```

class PopupAppMenu extends PopupMenu
    implements ActionListener {
    PopupApplication ref;
    public PopupAppMenu(PopupApplication ref) {
        super("File");
        this.ref = ref;
        MenuItem mi;
        add(mi = new MenuItem("Copy"));
        mi.addActionListener(this);
        add(mi = new MenuItem("Cut"));
        mi.addActionListener(this);
        add(mi = new MenuItem("Paste"));
    }
}

```

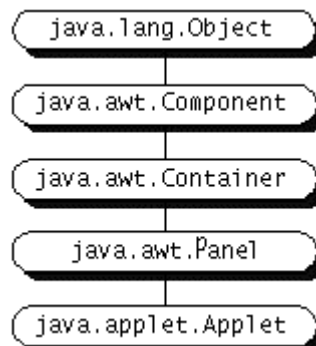
```
mi.addActionListener(this);  
}  
public void actionPerformed(ActionEvent e) {  
    String item = e.getActionCommand();  
    ref.msg.setText("Selected menu item: " + item);  
}  
}
```

## APPLET

### Overview of Applets

This lesson discusses the parts of an applet. If you haven't yet compiled an applet and included it in an HTML page, you might want to do so now.

Every applet is implemented by creating a subclass of the Applet class. The following figure shows the inheritance hierarchy of the Applet class. This hierarchy determines much of what an applet can do and how.





## A Simple Applet

Below is the source code for an applet called **Simple**. The **Simple** applet displays a descriptive string whenever it encounters a major milestone in its life, such as when the user first visits the page the applet's on. The pages that follow use the **Simple** applet and build upon it to illustrate concepts that are common to many applets.

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

public class Simple extends Applet {

    StringBuffer buffer;

    public void init() {
        buffer = new StringBuffer();
        addItem("initializing... ");
    }

    public void start() {
        addItem("starting... ");
    }
}
```

```
public void stop() {  
    addItem("stopping... ");  
}
```

```
public void destroy() {  
    addItem("preparing for unloading...");  
}
```

```
void addItem(String newWord) {  
    System.out.println(newWord);  
    buffer.append(newWord);  
    repaint();  
}
```

```
public void paint(Graphics g) {  
    //Draw a Rectangle around the applet's  
display area.
```

```
    g.drawRect(0, 0, size().width - 1,  
size().height - 1);
```

```
    //Draw the current string inside the  
rectangle.
```

```
    g.drawString(buffer.toString(), 5, 15);
```

```
    }  
}
```

## Loading the Applet

You should see "initializing... starting..." above, as the result of the applet being loaded. When an applet is loaded, here's what happens:

- An instance of the applet's controlling class (an `Applet` subclass) is created.
- The applet *initializes* itself.
- The applet *starts* running.

## Leaving and Returning to the Applet's Page

When the user leaves the page -- for example, to go to another page -- the applet has the option of *stopping* itself. When the user returns to the page, the applet can *start* itself again. The same sequence occurs when the user iconifies and then reopens the window that contains the applet. (Other terms used instead of *iconify* are *minaturize*, *minimize*, and *close*.)

## Reloading the Applet

Some browsers let the user reload applets, which consists of unloading the applet and then loading it again. Before an applet is unloaded, it's given the chance to *stop* itself and then to perform a *final cleanup*, so that the applet can release any resources it holds. After that, the applet is unloaded and then loaded again, as described above.

## Quitting the Browser

When the user quits the browser (or whatever application is displaying the applet), the applet has the chance to *stop* itself and do *final cleanup* before the browser exits.

## Summary

An applet can react to major events in the following ways:

- It can *initialize* itself.
- It can *start* running.
- It can *stop* running.
- It can perform a *final cleanup*, in preparation for being unloaded.

## Methods for Milestones

```
public class Simple extends Applet {  
    ...  
    public void init() { ... }  
    public void start() { ... }  
    public void stop() { ... }  
    public void destroy() { ... }  
    ...  
}
```

The **Simple** applet, like every other applet, features a subclass of the **Applet** class. The **Simple** class overrides four **Applet** methods so that it can respond to major events:

### init

To *initialize* the applet each time it's loaded (or reloaded).

### start

To *start* the applet's execution, such as when the applet's loaded or when the user revisits a page that contains the applet.

### stop

To *stop* the applet's execution, such as when the user leaves the applet's page or quits the browser.

### destroy

To perform a *final cleanup* in preparation for unloading.

Not every applet needs to override every one of these methods. Some very simple applets override none of them.

The `init` method is useful for one-time initialization that doesn't take very long. In general, the `init` method should contain the code that you would normally put into a constructor. The reason applets shouldn't usually have constructors is that an applet isn't guaranteed to have a full environment until its `init` method is called.

Every applet that does something after initialization (except in direct response to user actions) must override the `start` method. The `start` method either

performs the applet's work or (more likely) starts up one or more threads to perform the work.

Most applets that override `start` should also override the `stop` method. The `stop` method should suspend the applet's execution, so that it doesn't take up system resources when the user isn't viewing the applet's page. For example, an applet that displays animation should stop trying to draw the animation when the user isn't looking at it.

Many applets don't need to override the `destroy` method, since their `stop` method (which is called before `destroy`) does everything necessary to shut down the applet's execution. However, `destroy` is available for applets that need to release additional resources.

## **Methods for Drawing and Event Handling**

The `Simple` applet defines its onscreen appearance by overriding the `paint` method:

```
class Simple extends Applet {  
    . . .  
    public void paint(Graphics g) { . . . }  
    . . .  
}
```

```
}
```

The paint method is one of two display methods an applet can override:

### paint

The basic display method. Many applets implement the paint method to draw the applet's representation within a browser page.

### update

A method you can use along with paint to improve drawing performance.

Applets inherit their paint and update methods from the Applet class, which inherits them from the Abstract Window Toolkit (AWT) Component class. Applets inherit a group of event-handling methods from the Component class. To react to an event, an applet must override either the appropriate event-specific method.

## Methods for Adding UI Components

The Simple applet's display code (implemented in its paint method) is flawed: It doesn't support scrolling. Once the text it displays reaches the end of the display rectangle, you can't see any new text. Here's an example of the problem:

```
initializing... starting... stopping... starting... stopping... starting... stoppin
```

The simplest cure for this problem is to use a pre-made user interface (UI) component that has the right behavior.

## Pre-Made UI Components

The AWT supplies the following UI components (the class that implements each component is listed in parentheses):

- Buttons (`java.awt.Button`)
- Checkboxes (`java.awt.Checkbox`)
- Single-line text fields (`java.awt.TextField`)
- Larger text display and editing areas (`java.awt.TextArea`)
- Labels (`java.awt.Label`)
- Lists (`java.awt.List`)
- Pop-up lists of choices (`java.awt.Choice`)
- Sliders and scrollbars (`java.awt.Scrollbar`)
- Drawing areas (`java.awt.Canvas`)
- Menus (`java.awt.Menu`, `java.awt.MenuItem`, `java.awt.CheckboxMenuItem`)
- Containers (`java.awt.Panel`, `java.awt.Window` and its subclasses)



## Methods for Using UI Components in Applets

Because the `Applet` class inherits from the `AWT Container` class, it's easy to add components to applets and to use layout managers to control the components' onscreen positions.

Here are some of the `Container` methods an applet can use:

**add**

Adds the specified `Component`.

**remove**

Removes the specified `Component`.

**setLayout**

Sets the layout manager.

## Adding a Non-Editable Text Field to the Simple Applet

To make the `Simple` applet use a scrolling, non-editable text field, we can use the `TextField` class. Here is the revised [source code](#)♦. The changes are shown below.

```
//Importing java.awt.Graphics is no longer  
necessary  
//since this applet no longer implements the  
paint method.
```

```
...  
import java.awt.TextField;  
  
    public class ScrollingSimple extends  
Applet {  
  
        //Instead of using a StringBuffer, use a  
TextField:  
        TextField field;  
  
        public void init() {  
            //Create the text field and make it  
uneditable.  
            field = new TextField();  
            field.setEditable(false);  
  
            //Set the layout manager so that the text  
field will be  
as wide as possible.  
            setLayout(new  
java.awt.GridLayout(1,0));  
  
            //Add the text field to the applet.
```

```
        add(field);

        addItem("initializing... ");
    }

    ...

    void addItem(String newWord) {
        //This used to append the string to the
        StringBuffer;
        //now it appends it to the TextField.
        String t = field.getText();
        System.out.println(newWord);
        field.setText(t + newWord);
        repaint();
    }

    //The paint method is no longer necessary,
    //since the TextField repaints itself
    automatically.
```

The revised init method creates an uneditable text field (a TextField instance). It sets the applet's layout

manager to one that makes the text field as wide as possible and then adds the text field to the applet.

## Using the APPLET Tag

You should already have seen the simplest form of the <APPLET> tag:

```
<APPLET CODE=AppletSubclass.class
WIDTH=anInt HEIGHT=anInt>
</APPLET>
```

This tag tells the browser to load the applet whose Applet subclass is named *AppletSubclass*, displaying it in an area of the specified width and height.

## Specifying Parameters

Some applets let the user customize the applet's configuration with parameters.

The user specifies the value of a parameter using a <PARAM> tag. The <PARAM> tags should appear just after the <APPLET> tag for the applet they affect:

```
<APPLET CODE=AppletSubclass.class
WIDTH=anInt HEIGHT=anInt>
```

```
<PARAM NAME=parameter1Name
VALUE=aValue>
<PARAM NAME=parameter2Name
VALUE=anotherValue>
</APPLET>
```

Here's an example of the <PARAM> tag in use.

```
<APPLET CODE="Animator.class"
WIDTH=460 HEIGHT=160>
<PARAM NAME="imageSource"
VALUE="images/Beans">
<PARAM NAME="backgroundColor"
VALUE="0xc0c0c0">
<PARAM NAME="endImage" VALUE=10>
<PARAM NAME="soundSource"
VALUE="audio">
<PARAM NAME="soundtrack"
VALUE="spacemusic.au">
<PARAM NAME="sounds"
VALUE="1.au|2.au|3.au|4.au|5.au|6.au|7.au|8
au|9.au|0.au">
<PARAM NAME="pause" VALUE=200>
```

```
...  
</APPLET>
```

## Specifying Alternate HTML Code and Text

Note the ellipsis points (". . .") in the previous HTML example. What did the example leave out? It omitted *alternate HTML code* -- HTML code interpreted only by browsers that don't understand the `<APPLET>` tag. Alternate HTML code is any text that appears between the `<APPLET>` and `</APPLET>` tags, after any `<PARAM>` tags. Java-enabled browsers ignore alternate HTML code.

To specify alternate text to Java-enabled browsers and other browsers that understand the `<APPLET>` tag, use the ALT attribute. If the browser can't display an applet for some reason, it can display the applet's ALT text.

We use alternate HTML code throughout the online version of this tutorial to tell readers about the applets they're missing. Often, the alternate HTML code includes one or more pictures of the applet. Here's the complete HTML code for the Animator example shown previously:

```
<APPLET CODE="Animator.class"  
WIDTH=460 HEIGHT=160
```

```
ALT="If you could run this applet, you'd see
some animation">
<PARAM NAME="imageSource"
VALUE="images/Beans">
<PARAM NAME="backgroundColor"
VALUE="0xc0c0c0">
<PARAM NAME="endImage" VALUE=10>
<PARAM NAME="soundSource"
VALUE="audio">
<PARAM NAME="soundtrack"
VALUE="spacemusic.au">
<PARAM NAME="sounds"
VALUE="1.au|2.au|3.au|4.au|5.au|6.au|7.au|8
au|9.au|0.au">
<PARAM NAME="pause" VALUE=200>
Your browser is completely ignoring the
<APPLET> tag!
</APPLET>
```

An applet that doesn't understand the `<APPLET>` tag ignores everything in the previous HTML code except the line that starts with "Your". A browser that *does* understand the

`<APPLET>` tag ignores everything on that line. If the applet-savvy browser can't run the applet, it might display the ALT text.

## Specifying the Applet Directory

By default, a browser looks for an applet's class and archive files in the same directory as the HTML file that has the `<APPLET>` tag. (If the applet's class is in a package, then the browser uses the package name to construct a directory path underneath the HTML file's directory.) Sometimes, however, it's useful to put the applet's files somewhere else. You can use the `CODEBASE` attribute to tell the browser in which directory the applet's files are located:

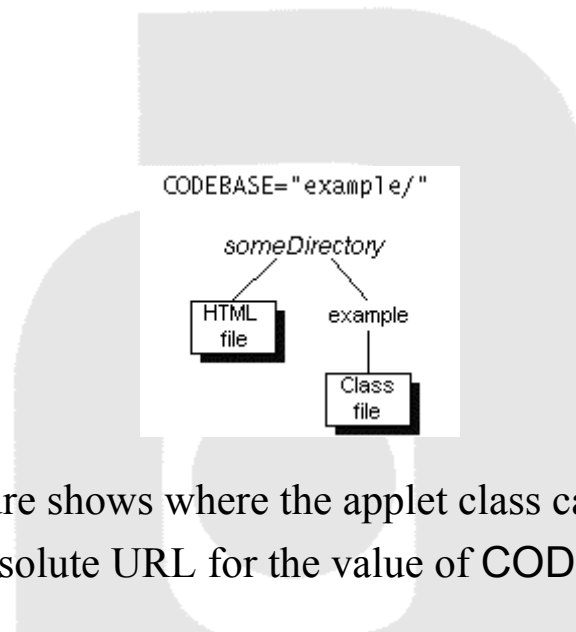
```
<APPLET CODE=AppletSubclass.class
CODEBASE=aURL
WIDTH=anInt HEIGHT=anInt>
</APPLET>
```

If *aURL* is a relative URL, then it's interpreted relative to the HTML document's location. By making *aURL* an absolute URL, you can load an applet from just about anywhere -- even from another HTTP server.

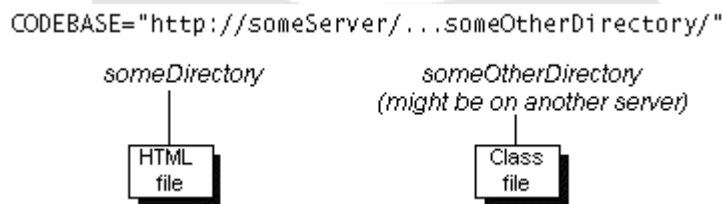


This tutorial uses  
`CODEBASE="someDirectory"` frequently,  
since we group the

The following figure shows the location of the class file,  
relative to the HTML file, when `CODEBASE` is set to  
"example/".



The next figure shows where the applet class can be if you  
specify an absolute URL for the value of `CODEBASE`.



## <APPLET> Tag Attributes

When you build <APPLET> tags, keep in mind that words such as `APPLET` and `CODEBASE` can be typed in either as shown or in any mixture of uppercase and lowercase. **Bold font** indicates something you should type in exactly as shown (except that letters don't need to be uppercase). *Italic font* indicates that you must substitute a value for the word in

italics. Square brackets ([ and ]) indicate that the contents of the brackets are optional.

### < APPLET

[CODEBASE = *codebaseURL*]

CODE = *appletFile*

[ALT = *alternateText*]

[NAME = *appletInstanceName*]

WIDTH = *pixels*

HEIGHT = *pixels*

[ALIGN = *alignment*]

[VSPACE = *pixels*]

[HSPACE = *pixels*]

>

[< PARAM NAME = *appletParameter1*

VALUE = *value* >]

[< PARAM NAME = *appletParameter2*

VALUE = *value* >]

...

[*alternateHTML*]

</APPLET>

**CODEBASE** = *codebaseURL*

This optional attribute specifies the base URL of the applet -- the directory or folder that contains the applet's code. If this attribute is not specified, then the document's URL is used.

**CODE** = *appletFile*

This required attribute gives the name of the file that contains the applet's compiled Applet subclass. This file is relative to the base URL of the applet. It cannot be absolute.

**ALT** = *alternateText*

This optional attribute specifies any text that should be displayed if the browser understands the APPLET tag but can't run Java applets.

**NAME** = *appletInstanceName*

This optional attribute specifies a name for the applet instance, which makes it possible for applets on the same page to find (and communicate with) each other.

**WIDTH** = *pixels*

**HEIGHT** = *pixels*

These required attributes give the initial width and height (in pixels) of the applet display area, not counting any windows or dialogs that the applet brings up.

**ALIGN** = *alignment*

This optional attribute specifies the alignment of the applet. The possible values of this attribute are the same (and have the same effects) as those for the IMG tag: left, right, top, texttop, middle, absmiddle, baseline, bottom, absbottom.

**VSPACE** = *pixels*

**HSPACE** = *pixels*

These optional attributes specify the number of pixels above and below the applet (VSPACE) and on each side of the applet (HSPACE). They're treated the same way as the IMG tag's VSPACE and HSPACE attributes.

**< PARAM NAME = *appletParameter1* VALUE = *value* >**

**<PARAM>** tags are the only way to specify applet-specific parameters. Applets read user-specified values for parameters with the `getParameter()` method. See [Defining and Using Applet Parameters](#) for information about the `getParameter()` method.

*alternateHTML*

If the HTML page containing this `<APPLET>` tag is viewed by a browser that doesn't understand the `<APPLET>` tag, then the browser will ignore the `<APPLET>` and `<PARAM>` tags, instead interpreting any other HTML code between the `<APPLET>` and `</APPLET>` tags. Java-compatible browsers ignore this extra HTML code.

## What Applets Can and Can't Do

Following is an overview of both the restrictions applets face and the special capabilities they have.

### Security Restriction

Every browser implements security policies to keep applets from compromising system security. This section describes the security policies that current browsers adhere to. However, the implementation of the security policies differs from browser to browser. Also, security policies are subject to change. For example, if a browser is developed for use only in trusted environments, then its security policies will likely be much more lax than those described here.

Current browsers impose the following restrictions on any applet that is loaded over the network:

- An applet cannot load libraries or define native methods.
- It cannot ordinarily read or write files on the host that's executing it.
- It cannot make network connections except to the host that it came from.

- It cannot start any program on the host that's executing it.
- It cannot read certain system properties.
- Windows that an applet brings up look different than windows that an application brings up.

Each browser has a **SecurityManager** object that implements its security policies. When a **SecurityManager** detects a violation, it throws a **SecurityException**. Your applet can catch this **SecurityException** and react appropriately.

### **Applet Capabilities**

The `java.applet` package provides an API that gives applets some capabilities that applications don't have. For example, applets can play sounds, which other programs can't do yet.

Here are some other things that current browsers and other applet viewers let applets do:

- Applets can usually make network connections to the host they came from.
- Applets running within a Web browser can easily cause HTML documents to be displayed.
- Applets can invoke public methods of other applets on the same page.
- Applets that are loaded from the local file system (from a directory in the user's **CLASSPATH**) have none of the restrictions that applets loaded over the network do.

## **Network Programming**

### **The Internet Protocol Suite**

The `java.net` package provides a set of classes that support network programming using the communication protocols employed by the Internet. These protocols are known as the *Internet protocol suite* and include the *Internet Protocol* (IP), the *Transport Control Protocol* (TCP), and the *User Datagram Protocol* (UDP) as well as other, less-prominent supporting protocols.

### **The Internet**

Asking the question What is the Internet? may bring about a heated discussion in some circles. In this book, the *Internet* is defined as the collection of all computers that are able to communicate, using the Internet protocol suite. This definition includes all computers to which you can directly (or indirectly through a firewall) send Internet Protocol packets.

Computers on the Internet communicate by exchanging packets of data, known as Internet Protocol, or IP, packets. IP is the network protocol used to send information from one computer to another over the Internet. All computers on the Internet (by our definition in this book) communicate using IP. IP moves information contained in IP packets. The IP packets are routed via special routing algorithms from a source computer that sends the packets to a destination computer that receives them. The routing

algorithms figure out the best way to send the packets from source to destination.

In order for IP to send packets from a source computer to a destination computer, it must have some way of identifying these computers. All computers on the Internet are identified using one or more IP addresses. A computer may have more than one IP address if it has more than one interface to computers that are connected to the Internet.

IP addresses are 32-bit numbers. They may be written in decimal, hexadecimal, or other formats, but the most common format is dotted decimal notation. This format breaks the 32-bit address up into four bytes and writes each byte of the address as unsigned decimal integers separated by dots. For example, 204.212.153.193.

IP addresses are not easy to remember, even using dotted decimal notation. The Internet has adopted a mechanism, referred to as the *Domain Name System* (DNS), whereby computer names can be associated with IP addresses. These computer names are referred to as *domain names*. The DNS has several rules that determine how domain names are constructed and how they relate to one another. For the purposes of this chapter, it is sufficient to know that domain names are computer names and that they are mapped to IP addresses.

The mapping of domain names to IP addresses is maintained by a system of *domain name servers*. These servers are able to look up the IP address corresponding to a domain name. They also provide the capability to look up the domain name associated with a particular IP address, if one exists.

IP enables communication between computers on the Internet by routing data from a source computer to a destination computer. However, computer-to-computer communication only solves half of the network communication problem. In order for an application program, such as a mail program, to communicate with another application, such as a mail server, there needs to be a way to send data to specific programs within a computer.

Ports are used to enable communication between programs. A *port* is an address within a computer. Port addresses are 16-bit addresses that are usually associated with a particular application protocol. An application server, such as a Web server or an FTP server, listens on a particular port for service requests, performs whatever service is requested of it, and returns information to the port used by the application program requesting the service.

Popular Internet application protocols are associated with *well-known ports*. The server programs implementing these protocols listen on these ports for service requests. The well-known ports for some common Internet application protocols are



## Port Protocol

- 21 File Transfer Protocol
- 23 Telnet Protocol
- 25 Simple Mail Transfer Protocol
- 80 Hypertext Transfer Protocol

The well-known ports are used to standardize the location of Internet services.

### **Connection-Oriented Versus Connectionless Communication**

Transport protocols are used to deliver information from one port to another and thereby enable communication between application programs. They use either a connection-oriented or connectionless method of communication. TCP is a connection-oriented protocol and UDP is a connectionless transport protocol. The TCP connection-oriented protocol establishes a communication link between a source port/IP address and a destination port/IP address. The ports are bound together via this link until the connection is terminated and the link is broken. An example of a connection-oriented protocol is a telephone conversation. A telephone connection is established, communication takes place, and then the connection is terminated.

The reliability of the communication between the source and destination programs is ensured through error-detection and

error-correction mechanisms that are implemented within TCP. TCP implements the connection as a stream of bytes from source to destination. This feature allows the use of the stream I/O classes provided by java.io.

The UDP connectionless protocol differs from the TCP connection-oriented protocol in that it does not establish a link for the duration of the connection. An example of a connectionless protocol is postal mail. To mail something, you just write down a destination address (and an optional return address) on the envelope of the item you're sending and drop it in a mailbox. When using UDP, an application program writes the destination port and IP address on a datagram and then sends the datagram to its destination. UDP is less reliable than TCP because there are no delivery-assurance or error-detection and -correction mechanisms built into the protocol.

Application protocols such as FTP, SMTP, and HTTP use TCP to provide reliable, stream-based communication between client and server programs. Other protocols, such as the Time Protocol, use UDP because speed of delivery is more important than end-to-end reliability.

### **Sockets and Client/Server Communication**

Clients and servers establish connections and communicate via *sockets*. Connections are communication links that are created

over the Internet using TCP. Some client/server applications are also built around the connectionless UDP. These applications also use sockets to communicate.

Sockets are the endpoints of Internet communication. Clients create client sockets and connect them to server sockets. Sockets are associated with a host address and a port address. The host address is the IP address of the host where the client or server program is located. The port address is the communication port used by the client or server program. Server programs use the well-known port number associated with their application protocol. A client communicates with a server by establishing a connection to the socket of the server. The client and server then exchange data over the connection. Connection-oriented communication is more reliable than connectionless communication because the underlying TCP provides message-acknowledgment, error-detection, and error-recovery services.

When a connectionless protocol is used, the client and server communicate by sending datagrams to each other's socket. The UDP is used for connectionless protocols. It does not support reliable communication like TCP.

## Overview of java.net

The java.net package provides several classes that support socket-based client/server communication.

The InetAddress class encapsulates Internet IP addresses and supports conversion between dotted decimal addresses and hostnames. The Socket, ServerSocket, and DatagramSocket classes implement client and server sockets for connection-oriented and connectionless communication. The URL, URLConnection, and URLEncoder classes implement high-level browser-server Web connections.

### The InetAddress Class

The InetAddress class encapsulates Internet addresses. It supports both numeric IP addresses and hostnames.

The InetAddress class has no public variables or constructors. It provides eight access methods that support common operations on Internet addresses. Three of these methods are static.

The getLocalHost() method is a static method that returns an InetAddress object representing the Internet address of the local host computer. The static getByName() method returns an

InetAddress object for a specified host. The static `getAllByName()` method returns an array of all Internet addresses associated with a particular host.

The `getAddress()` method gets the numeric IP address of the host identified by the InetAddress object, and the `getHostName()` method gets its domain name.

The `equals()`, `hashCode()`, and `toString()` methods override those of the Object class.

The NSLookupApp program illustrates the use of the InetAddress class. It takes a hostname as a parameter and identifies the primary IP address associated with that host.

```
import java.net.*;

public class NSLookupApp {
    public static void main(String args[]) {
        try {
            if(args.length!=1){
                System.out.println("Usage: java NSLookupApp
hostName");
                return;
            }
            InetAddress host = InetAddress.getByName(args[0]);
            System.out.println("Host name: "+
```

```
host.getHostName());
    System.out.print("IP address: "+
host.getHostAddress());
}catch(UnknownHostException ex) {
    System.out.println("Unknown host");
return;
}
}
}
```

This code example uses NSLookupApp to look up the primary IP address associated with the host.

### **The Socket Class**

The Socket class implements client connection-based sockets.

These sockets are used to develop applications that utilize services provided by connection-oriented server applications.

The Socket class provides four constructors that create sockets and connect them to a destination host and port. The access methods are used to access the I/O streams and connection parameters associated with a connected socket.

The getInetAddress() and getPort() methods get the IP address of the destination host and the destination host port number to which the socket is connected. The getLocalPort() method returns the

source host local port number associated with the socket. The `getInputStream()` and `getOutputStream()` methods are used to access the input and output streams associated with a socket. The `close()` method is used to close a socket.

The `setSocketImplFactory()` class method is used to switch from the default Java socket implementation to a custom socket implementation.

The following program is used to talk to a particular port on a given host on a line-by-line basis. It provides the option of sending a line to the specified port, receiving a line from the other host, or terminating the connection.

```
/*Client*/
import java.net.*;
import java.io.*;

public class Client
{
    public static void main(String args[]) {
        Socket socket;

        try {
            DataInputStream input;
            PrintStream output;
```

```
socket = new Socket("localhost", 10000);
while( true ) {
    try {
        String tmp;
        DataInputStream user_input = new
DataInputStream(System.in);

        input = new DataInputStream(socket.getInputStream());
        output = new PrintStream(socket.getOutputStream());
        tmp = user_input.readLine();
        output.println(tmp);
        System.out.println(tmp);
        tmp = input.readLine();
        System.out.println(tmp);
    }
    catch( java.io.IOException e ) {
        e.printStackTrace();
        return;
    }
}
catch( java.io.IOException e ) {
```



```
        e.printStackTrace();
    }
}
}
```

## **The ServerSocket Class**

The ServerSocket class implements a TCP server socket. It provides two constructors that specify the port to which the server socket is to listen for incoming connection requests. An optional count parameter may be supplied to specify the amount of time that the socket should listen for an incoming connection.

The accept() method is used to cause the server socket to listen and wait until an incoming connection is established. It returns an object of class Socket once a connection is made. This Socket object is then used to carry out a service for a single client. The getInetAddress() method returns the address of the host to which the socket is connected. The getLocalPort() method returns the port on which the server socket listens for an incoming connection. The toString() method returns the socket's address and port number as a string in preparation for printing.

The close() method closes the server socket.

The static setSocketFactory() method is used to change the default ServerSocket implementation to a custom implementation.

```
/*Server*/
```

```
import java.net.*;
```

```
import java.io.*;
```

```
public class Server implements Runnable
```

```
{
```

```
    private Socket client;
```

```
    public Server(Socket socket)
```

```
    {
```

```
        Thread thread;
```

```
        client = socket;
```

```
        thread = new Thread(this);
```

```
        thread.start();
```

```
    }
```

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

```
    {
```

```
        ServerSocket listen_socket;
```

```
        try
```

```
        {
```

```
            listen_socket = new ServerSocket(10000);
```

```
    }  
    catch( java.io.IOException e )  
    {  
        System.err.println("Failed to create listen socket.");  
        e.printStackTrace();  
        System.exit(-1);  
        return;  
    }  
    while( true )  
    {  
        Socket socket;  
        try  
        {  
            socket = listen_socket.accept();  
            new Server(socket);  
        }  
        catch( java.io.IOException e )  
        {  
            e.printStackTrace();  
        }  
    }  
}  
  
public void run()
```

```
{
    DataInputStream input;
    PrintStream output;
    String data;
    try
    {
        input = new
DataInputStream(client.getInputStream());
        output = new
PrintStream(client.getOutputStream());
    }
    catch( java.io.IOException e )
    {
        e.printStackTrace();
        return;
    }
    while( true )
    {
        try
        {
            data = input.readLine();
            output.println("Received (" + data.length() +
"); " + data);
```

```
        System.out.println(data);
    }
    catch( java.io.IOException e )
    {
        break;
    }
}
}
```

### **The DatagramSocket Class**

The DatagramSocket class is used to implement client and server sockets using the UDP protocol. UDP is a connectionless protocol that allows application programs (both clients and servers) to exchange information using chunks of data known as *datagrams*.

DatagramSocket provides two constructors. The default constructor creates a datagram socket for use by client applications. No port number is specified. The second constructor allows a datagram socket to be created using a specified port. This constructor is typically used with server applications.

The send() and receive() methods are used to send and receive datagrams using the socket. The datagrams are objects of class DatagramPacket. The getLocalPort() method returns the local

port used in the socket. The `close()` method closes this socket, and the `finalize()` method performs additional socket-termination processing when the socket is deallocated during garbage collection.

### **The DatagramPacket Class**

The `DatagramPacket` class encapsulates the actual datagrams that are sent and received using objects of class `DatagramSocket`. Two different constructors are provided: one for datagrams that are received from a datagram socket and one for creating datagrams that are sent over a datagram socket. The arguments to the received datagram constructor are a byte array used as a buffer for the received data and an integer that identifies the number of bytes received and stored in the buffer. The sending datagram constructor adds two additional parameters: the IP address and port where the datagram is to be sent.

Four access methods are provided. The `getAddress()` and `getPort()` methods are used to read the destination IP address and port of the datagram. The `getLength()` and `getData()` methods are used to get the number of bytes of data contained in the datagram and to read the data into a byte array buffer.

```
/*DatagramSender*/
```

```
import java.net.*;

class DatagramSender {

    public static void main(String args[]) {
        try {

            // Create destination Internet address
            InetAddress ia =
                InetAddress.getByName(args[0]);

            // Obtain destination port
            int port = Integer.parseInt(args[1]);

            // Create a datagram socket
            DatagramSocket ds = new DatagramSocket();

            // Create a datagram packet
            byte buffer[] = args[2].getBytes();
            DatagramPacket dp =
                new DatagramPacket(buffer, buffer.length,
                    ia, port);
```

```
// Send the datagram packet
ds.send(dp);
}
catch(Exception e) {
    e.printStackTrace();
}
}
}

/*DatagramReceiver */

import java.net.*;

class DatagramReceiver {
    private final static int BUFSIZE = 20;

    public static void main(String args[]) {
        try {

            // Obtain port
            int port = Integer.parseInt(args[0]);
```



```
// Create a DatagramSocket object for the port
DatagramSocket ds = new DatagramSocket(port);

// Create a buffer to hold incoming data
byte buffer[] = new byte[BUFSIZE];

// Create infinite loop
while(true) {

    // Create a datagram packet
    DatagramPacket dp =
        new DatagramPacket(buffer, buffer.length);

    // Receive data
    ds.receive(dp);

    // Get data from the datagram packet
    String str = new String(dp.getData());

    // Display the data
    System.out.println(str);
}
}
```

```
    catch(Exception e) {  
        e.printStackTrace();  
    }  
}  
}
```

## **URL**

The URL class encapsulates Web objects by their URL address. It provides a set of constructors that allow URL objects to be easily constructed and a set of access methods that allow high-level read and write operations to be performed using URLs. The URL access methods provide a full set of URL processing capabilities. The `getProtocol()`, `getHost()`, `getPort()`, `getFile()`, and `getRef()` methods allow the individual address components of the URL to be determined. The `getContent()` and `openStream()` methods allow reading of the Web object pointed to by the URL. The `toExternalForm()` and `toString()` methods enable URLs to be converted into strings to support display and printing. The `equals()` method compares URLs, and the `sameFile()` method compares the Web objects pointed to by the URLs. The `GetURLApp` program illustrates the power provided by the URL class. This small program implements a primitive Web browser. Just run the program with the name of an URL and it

makes a connection to the destination Web server and downloads the referenced document.

```
import java.net.*;
import java.io.*;

public class GetURLApp {
    public static void main(String args[]){
        try{
            if(args.length!=1) error("Usage: java GetURLApp
URL");
            System.out.println("Fetching URL: "+args[0]);
            URL url = new URL(args[0]);
            DataInputStream inStream = new
DataInputStream(url.openStream());
            String line;
            while ((line = inStream.readLine())!= null){
                System.out.println(line);
            }
            inStream.close();
        }catch (MalformedURLException ex){
            error("Bad URL");
        }catch (IOException ex){
            error("IOException occurred.");
        }
    }
}
```

```
    }  
    }  
    public static void error(String s){  
        System.out.println(s);  
        System.exit(1);  
    }  
}
```

---

*”Be fluent at your thoughts, express it, anyway...”*

*- Bhavin Rawal*

