1

INTRODUCTION TO JAVA PROGRAMMING

Unit Structure

1.1OBJECT ORIENTED PROGRAMMING REVISITED

Introduction to OOPS:

OOPS acronym stands for Object-oriented programming structure. This can be characterized as data controlling access to code.

This model is based on three concepts called, Encapsulation, Inheritance and polymorphism.

Encapsulation:

Encapsulation provides the ability to hide the internal details of an object from its users. Users many not be able to change the state of an object directly. But the state of an object can be changed by methods. It is also called data hiding or information hiding. Data and methods operating on the data are wrapped in a single object.

Inheritance:

Inheritance facilitates the reusability of existing code by building new classes using the definition of the existing classes.

Polymorphism:

Polymorphism is the third concept of OOP. It is the ability to take more than one form. One operation may exhibit different behavior in different situations. For example, an addition operation involving two numbers will produce a number the sum, but the same addition will produce a string if operands are strings.

The ability to redefine a routine in a derived class is called polymorphism.

For example we can have a method called area defined in a shape class which can be redefined in circle and square using different formula. When the shape area is called with circle you get area of circle. And with square you get area of square.

Explanation of the statement: “Java: A simple, Object-oriented, network-savvy, interpreted, robust secure, architecture-neutral, portable, high performance, multithreaded dynamic language”.The fundamental forces that necessitated the invention of java are portability and security; other factors also played an important role.

Java was designed to be easy for the professional programmer to learn and use effectively. If you have some programming experience you will not find java hard to master. If you already know the basic concepts of object-oriented programming, learning java will be even easier. Because java inherits c/c++ syntax and object oriented features of c++, most programmers have little trouble learning java. Some confusing concepts from c++, such as pointers left out of java or implemented in a cleaner, more approachable manner.Also makes an effort not to have surprising features. There are a small number of clearly defined ways to accomplish a given task.

Object-Oriented:

Java is an object-oriented language. It got a clean, usable pragmatic approach to objects. Java strikes a balance between the two thoughts, “every thing should be an object” and “stay out of objects”. The object model in java is simple easy to extend, while simple types, such as integers, are kept as high-performance non-objects.

Network-savvy:

Java is platform-independent. The Internet helped java to the forefront of programming. The reason is java expands of the universe of objects that can move about freely in cyberspace. In a network, two categories of objects are transmitted between the server and client. If you read your mail it is only static data. Even if you download a program code, is static till you execute it. The second type is a dynamic self-executing program. Such a program is an active agent on the client computer, yet is initiated by the server. For example, a program might be provided by the server to display properly, the data that the server is sending. In addition to being dynamic, network programs present problems in areas of security and portability. Java addresses these concerns with applets.

Java handles TCP/IP protocols for Internet. Accessing a URL is not much different from accessing a file. Intra address space messaging allowed objects on two different computers to execute procedures remotely. Java revived these interfaces in a package called Remote Method Invocation (RMI). This feature brings an unparallel level of abstraction to client/server performance.

Interpreted and high performance:

Java enables the creation of cross—platform programs by compiling into an intermediate representation called java byte code. This code can be interpreted on any system that provides a java virtual machine. Most of the previous interpreted programs like PERL; TEL etc suffer from almost insurmountable performance deficits. Even though java uses Byte code to be interpreted, it is designed to translate into native machine code for very high-performance by using just-in-time compiler.

Robust:

The multiplatform environment of the web requires the programs to execute reliably in a variety of systems. Thus the ability to make robust programs is given high priority in design of java. Java forces you to find mistakes early in program development. Java checks your code at compile time and run time. Many hard to track down bugs that often turn up at run time is impossible in java. Java does the memory management by itself by allotting memory and clearing garbage. The deallocation of memory is almost automatic. The division by zero, and file not found errors, are managed by object oriented exception handling.

Secure:

Every time you download a normal program, you are risking a viral infection. In addition to virus, another malicious program exists, which gather private information, by searching your computers local file system. When you use a java compatible web browser, you can safely download java applets without fear of viral infection or malicious intent. Java confines the programs to java execution environment and do not allow it access to other parts of the computer. The ability to download applets with confidence that no harm will be done and that no security will be breached is considered by many to be single most important aspect of java.

Architecture-neutral:

A central issue for designers is that of code longevity and portability. There is no guarantee exists that if you write a program today, it will run tomorrow—even on the same machine. OS upgrades, processor upgrades, and changes in core system resources can all combine to make a program malfunction. The java language and java virtual machine, the goal was “write once; run anywhere, any time, and forever.” To a great extent, this goal was achieved.

Portable:

Many types of computers and OS are in use throughout the world and many are connected to the Internet. For programs to be dynamically downloaded to all the various types of platforms connected to the Internet, some means of generating portable executable code is needed. Java’s byte code is portable.

Multithreaded:

Java was designed to meet the real-world requirement of creating interactive, networked programs. Java supports multithreaded programming, which allows you to write programs that do many things simultaneously. Java’s easy to use approach to multithreading allows to you to think about specific behavior of your program, not the multitasking subsystem.

Dynamic:

Java programs carry with them substantial amounts of run time type information that is used to verify and resolve accesses to objects at run time. This makes it possible to dynamically link code in a safe and expedient manner. This is crucial to the robustness to the applet environment, in which small fragments of byte code may be dynamically updated on a running system.

1.2 JDK, JAVA VIRTUAL MACHINE

The Java Development Kit (JDK) is an implementation of either one of the Java SE, Java EE or Java MEplatforms

The JDK has as its primary components a collection of programming tools, including:

appletviewer – this tool can be used to run and debug Java applets without a web browser

apt – the annotation-processing tool

extcheck – a utility which can detect JAR-file conflicts

idlj – the IDL-to-Java compiler. This utility generates Java bindings from a given Java IDL file.

java – the loader for Java applications. This tool is an interpreter and can interpret the class files generated by the javac compiler. Now a single launcher is used for both development and deployment. The old deployment launcher, jre, no longer comes with Sun JDK, and instead it has been replaced by this new java loader.

javac – the Java compiler, which converts source code into Java bytecode

javadoc – the documentation generator, which automatically generates documentation from source code comments

jar – the archiver, which packages related class libraries into a single JAR file. This tool also helps manage JAR files.

javah – the C header and stub generator, used to write native methods

javap – the class file disassembler

javaws – the Java Web Start launcher for JNLP applications

JConsole – Java Monitoring and Management Console

jdb – the debugger

jhat – Java Heap Analysis Tool (experimental)

jinfo – This utility gets configuration information from a running Java process or crash dump. (experimental)

jmap – This utility outputs the memory map for Java and can print shared object memory maps or heap memory details of a given process or core dump. (experimental)

jps – Java Virtual Machine Process Status Tool lists the instrumented HotSpot Java Virtual Machines (JVMs) on the target system. (experimental)

jrunscript – Java command-line script shell.

jstack – utility which prints Java stack traces of Java threads (experimental)

jstat – Java Virtual Machine statistics monitoring tool (experimental)

jstatd – jstat daemon (experimental)

keytool – tool for manipulating the keystore

pack200 – JAR compression tool

policytool – the policy creation and management tool, which can determine policy for a Java runtime, specifying which permissions are available for code from various sources

VisualVM – visual tool integrating several command-line JDK toolsandlightweight performance and memoryprofiling capabilities

wsimport – generates portable JAX-WS artifacts for invoking a web service.

xjc – Part of the Java API for XML Binding (JAXB) API. It accepts an XML schema and generates Java classes.

Java Virtual machine:

Java Virtual Machine, or JVM as its name suggest is a “virtual” computer that resides in the real” computer as a software process. JVM gives Java the flexibility of platform independence. Let us see first how exactly Java program is created, compiled and executed.Java code is written in .java file. This code contains one or more Java language attributes like Classes, Methods, Variable, Objects etc. Javac is used to compile this code and to generate .class file. Class file is also known as “**byte code**“. The name byte code is given may be because of the structure of the instruction set of Java program. Java byte code is an input to Java Virtual Machine. JVM read this code and interpret it and executes the program.

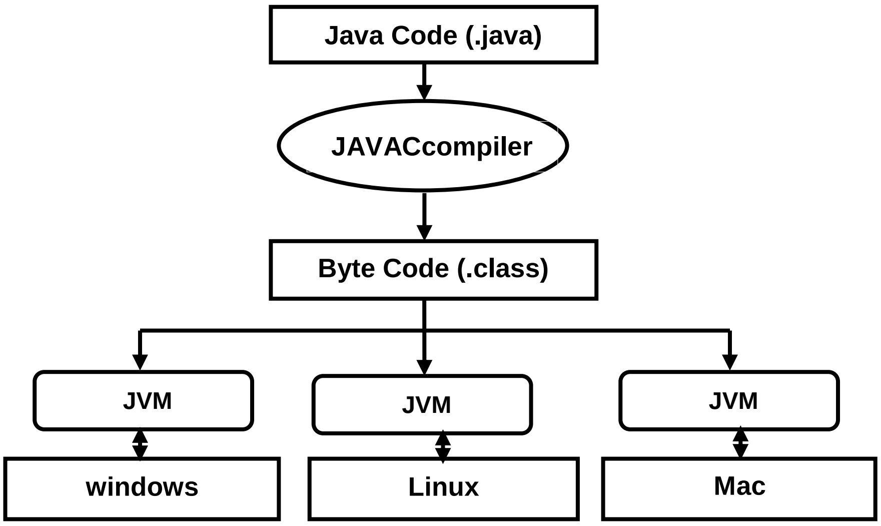
Why java is platform independent?:

Java solves the problem of platform-independence by usingbyte code. The Java compiler does not produce nativeexecutable code for a particular machine like a C compilerwould. Instead it produces a special format called bytecode. Java byte code written in hexadecimal, byte by byte,looks like this

CA FE BA BE 00 03 00 2D 00 3E 08 00 3B 08 00 01 08 00 20 08

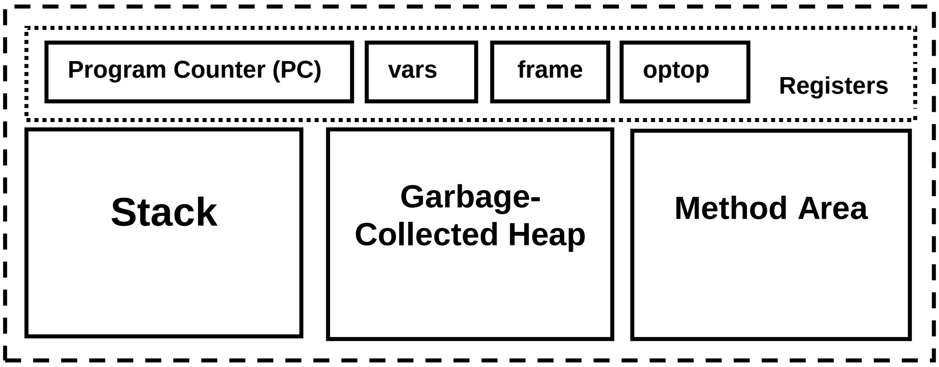
This looks a lot like machine language, but unlike machinelanguage Java byte code is exactly the same on everyplatform. This byte code fragment means the same thing on aSolaris workstation as it does on a Macintosh PowerBook.Java programs that have been compiled into byte code stillneed an interpreter to execute them on any given platform.The interpreter reads the byte code and translates it intothe native language of the host machine on the fly. The mostcommon such interpreter is Sun's program java (with a littlej). Since the byte code is completely platform independent,only the interpreter and a few native libraries need to beported to get Java to run on a new computer or operatingsystem. The rest of the runtime environment including thecompiler and most of the class libraries are written in Java.All these pieces, the javac compiler, the java interpreter,the Java programming language, and more are collectivelyreferred to as Java.

Fig. 1.1



Java Virtual Machine:

Java Virtual Machine like its real counter part, **executes** the program and generate output. To execute any code, JVM utilizes different components.JVM is divided into several components like the stack, the garbage-collected heap, the registers and the method area. Let us see diagram representation of JVM.



The Stack:

Stack in Java virtual machine stores various method arguements as well as the local variables of any method. Stack also keep track of each an every method invocation. This is called **Stack Frame**. There are three registers thats help in stack manipulation. They are vars, frame, optop. This registers points to different parts of current Stack.

There are three sections in Java stack frame:

Local Variables:

The local variables section contains all the local variables being used by the current method invocation. It is pointed to by the **vars** register.

Execution Environment:

The execution environment section is used to maintain the operations of the stack itself. It is pointed to by the **frame** register.

Operand Stack :

The operand stack is used as a work space by bytecode instructions. It is here that the parameters for bytecode instructions are placed, and results of bytecode instructions are found. The top of the operand stack is pointed to by the **optop** register.

Method Area:

This is the area where bytecodes reside. The program counter points to some byte in the method area. It always keep tracks of the current instruction which is being executed (interpreted). After execution of an instruction, the JVM sets the PC to next instruction. Method area is shared among all the threads of a process. Hence if more then one threads are accessing any specific method or any instructions, synchorization is needed. Synchronization in JVM is acheived through Monitors.

Garbage-collected Heap:

The Garbage-collected Heap is where the objects in Java programs are stored. Whenever we allocate an object using new operator, the heap comes into picture and memory is allocated from there. Unlike C++, Java does not have free operator to free any previously allocated memory. Java does this automatically using Garbage collection mechanism. Till Java 6.0, **mark and sweep** algorithm is used as a garbage collection logic. Remember that the local object reference resides on Stack but the actual object resides in Heap only. Also, arrays in Java are objects, hence they also resides in Garbage-collected Heap

1.3 PLATFORM INDEPENDENCE PORTABILITY-SCALABILITY

Java ensures portability in two ways. First, Java Compiler generates bytecodeinstructstions that can be implemented on any machine. Secondly, the size of the primitive data types are machine independent.

In the world of software, scalability refers to an application's ability to adapt to changing demands successfully. Java, a programming language, possesses several valuable features that allow Java applications to scale efficiently.

Scalability, Multithreading, Network Scalability and Swing Toolkit

1.4 OPERATORS AND EXPRESSION-DECISION MAKING

Java provides a rich operator environment. Most of its operators can be divided into the following four groups: arithmetic, bitwise, relational, and logical. Java also defines some additional operators that handle certain special situations.
Arithmetic Operators

Arithmetic operators are used in mathematical expressions in the same way that they are used in algebra. The following table lists the arithmetic operators:

Tabel 1.1

Operator	**Result**
+	Addition
-	Subtraction
*	Multiplication
/	Division
%	Modulus
++	Increment
+=	Addition
-=	Subtraction
*=	Multiplication
/=	Division
%=	Modulus
--	Decrement

The operands of the arithmetic operators must be of a numeric type. You cannot usethem on boolean types, but you can use them on char types, since the char type in Java is, essentially, a subset of int.

The Basic Arithmetic Operators :

The basic arithmetic operations-addition, subtraction, multiplication, and division- all behave as you would expect for all numeric types. The minus operator also has a unary form which negates its single operand. Remember that when the division operator is applied to an integer type, there will be no fractional component attached to the result.

The following simple example program demonstrates the arithmetic operators. It also illustrates the difference between floating-point division and integer division.

// Demonstrate the basic arithmetic operators.

```
classBasicMath {
public static void main(String args[])
 {
// arithmetic using integers System.out.println("Integer Arithmetic");
int a = 1 + 1;
int b = a * 3; int c = b / 4; int d = c - a; int e = -d;
System.out.println("a = " + a);
System.out.println("b = " + b);
System.out.println("c = " + c); System.out.println("d = " + d);
System.out.println("e = " + e);
// arithmetic using doubles System.out.println("\\nFloating Point Arithmetic");
 double da = 1 + 1;
doubledb = da * 3;
double dc = db / 4;
double dd = dc - a;
double de = -dd;
System.out.println("da = " + da);
System.out.println("db = " + db);
System.out.println("dc = " + dc);
System.out.println("dd = " + dd);
System.out.println("de = " + de);
}
}
```

When you run this program, you will see the following output:

```
Integer Arithmetic a = 2
b = 6 c = 1
d = -1 e = 1
Floating Point Arithmetic da = 2
db = 6
dc = 1.5
dd = -0.5 de = 0.5
```

The Modulus Operator:

The modulus operator, %, returns the remainder of a division operation. It can be applied to floating-point types as well as integer types. (This differs from C/C++, in which the % can only be applied to integer types.) The following example program demonstrates the

%:

```
// Demonstrate the % operator.
class Modulus {
public static void main(String args[]) {
int x = 42;
double y = 42.3;

System.out.println("x mod 10 = " + x % 10);
System.out.println("y mod 10 = " + y % 10);
}
}
```

When you run this program you will get the following output:

```
x mod 10 = 2
y mod 10 = 2.3
```

Arithmetic Assignment Operators:

Java provides special operators that can be used to combine an arithmetic operation with an assignment. As you probably know, statements like the following are quite common in programming:

```
a = a + 4;
```

In Java, you can rewrite this statement as shown here:

```
a += 4;
```

This version uses the += assignment operator. Both statements perform the same action:
they increase the value of a by 4. Here is another example,

```
a = a % 2;
```

which can be expressed as

```
a %= 2;
```

In this case, the %= obtains the remainder of a/2 and puts that result back into a. There are assignment operators for all of the arithmetic, binary operators. Thus, anystatement of the form

```
var = var op expression;
```

can be rewritten as

```
var op= expression;
```

The assignment operators provide two benefits. First, they save you a bit of typing, because they are "shorthand" for their equivalent long forms. Second, they are implemented more efficiently by the Java run-time system than are their equivalent long forms. For these reasons, you will often see the assignment operators used in professionally written Java programs.

Here is a sample program that shows several op= operator assignments in action:

```
// Demonstrate several assignment operators. classOpEquals {
public static void main(String args[]) {
int a = 1;
int b = 2;
int c = 3;
a += 5;
b *= 4;
c += a * b;
c %= 6;
System.out.println("a = " + a);
System.out.println("b = " + b);
System.out.println("c = " + c);
}
}
```

The output of this program is shown here:

```
a = 6
b = 8
c = 3
```

Increment and Decrement :

The ++ and the - - are Java's increment and decrement operators. they have some special properties that make them quite interesting.

The increment operator increases its operand by one. The decrement operator decreases its operand by one. For example, this statement:

x = x + 1;
can be rewritten like this by use of the increment operator:
x++;

Similarly, this statement:
x = x - 1;
is equivalent to
x--;

These operators are unique in that they can appear both in postfix form, where they follow the operand as just shown, and prefix form, where they precede the operand. In the foregoing examples, there is no difference between the prefix and postfix forms. However, when the increment and/or decrement operators are part of a larger expression, then a subtle, yet powerful, difference between these two forms appears. In the prefix form, the operand is incremented or decremented before the value is obtained for use in the expression. In postfix form, the previous value is obtained for use in the expression, and then the operand is modified. For example:

```
x = 42;
y = ++x;
```

In this case, y is set to 43 as you would expect, because the increment occurs before x is assigned to y. Thus, the line y = ++x; is the equivalent of these two statements:

```
x = x + 1;
y = x;
```

However, when written like this,

```
x = 42;
y = x++;
```

the value of x is obtained before the increment operator is executed, so the value of y is
42. Of course, in both cases x is set to 43. Here, the line y = x++; is the equivalent of these two statements:

```
y = x;
x = x + 1;
```

The following program demonstrates the increment operator.

```
// Demonstrate ++. classIncDec {
public static void main(String args[]) {
int a = 1;
 int b = 2;
 int c;
int d;
c = ++b;
d = a++;
 c++;
System.out.println("a = " + a);
System.out.println("b = " + b);
 System.out.println("c = " + c);
 System.out.println("d = " + d);
}
}
```

The output of this program follows:

```
a = 2
```

b = 3
c = 4
d = 1

The Bitwise Operators:

Java defines several bitwise operators which can be applied to the integer types, long, int, short, char, and byte. These operators act upon the individual bits of their operands. They are summarized in the table 1.2:

Tabel 1.2

Operator	Result
~	Bitwise unary NOT
&	Bitwise AND
|	Bitwise OR
^	Bitwise exclusive OR
>>	Shift right
>>>	Shift right zero fill
<<	Shift left
&=	Bitwise AND assignment
|=	Bitwise OR assignment
^=	Bitwise exclusive OR assignment
>>=	Shift right assignment
>>>=	Shift right zero fill assignment
<<=	Shift left assignment

Since the bitwise operators manipulate the bits within an integer, it is important to understand what effects such manipulations may have on a value. Specifically, it is useful to know how Java stores integer values and how it represents negative numbers. So, before continuing, let's briefly review these two topics.

All of the integer types are represented by binary numbers of varying bit widths. For example, the byte value for 42 in binary is 00101010, where each position represents a power of two, starting with 2at the rightmost bit. The next bit position to the left would be21, or 2, continuing toward the left with 22, or 4, then 8, 16, 32, and so on. So 42 has 1 bits set at positions 1, 3, and 5 (counting from 0 at the right); thus 42 is the sum of 21 +23 + 25, which is 2 + 8 + 32.

All of the integer types (except char) are signed integers. This means that they can represent negative values as well as positive ones. Java uses an encoding known as two's complement, which means that negative numbers are represented by inverting (changing 1's to 0's and vice versa) all of the bits in a value, then adding 1 to the result. For example, -42 is represented by inverting all of the bits in 42, or 00101010, which yields 11010101, then adding 1, which results in 11010110, or -42. To decode a negative number, first invert all of the bits, then add 1. -42, or 11010110 inverted yields 00101001, or 41, so when you add 1 you get 42.

The reason Java (and most other computer languages) uses two's complement is easy to see when you consider the issue of zero crossing. Assuming a byte value, zero isrepresented by 00000000. In one's complement, simply inverting all of the bits creates

11111111, which creates negative zero. The trouble is that negative zero is invalid in integer math. This problem is solved by using two's complement to represent negative values. When using two's complement, 1 is added to the complement, producing100000000. This produces a 1 bit too far to the left to fit back into the byte value, resulting in the desired behavior, where -0 is the same as 0, and 11111111 is the encoding for -1. Although we used a byte value in the preceding example, the same basic principle applies to all of Java's integer types.

Because Java uses two's complement to store negative numbers-and because all integers are signed values in Java-applying the bitwise operators can easily produce unexpected results. For example, turning on the high-order bit will cause the resulting value to be interpreted as a negative number, whether this is what you intended or not. To avoid unpleasant surprises, just remember that the high-order bit determines the sign of an integer no matter how that high-order bit gets set.

The Bitwise Logical Operators :

The bitwise logical operators are &, |, ^, and ~. The following table shows the outcome of each operation. In the discussion that follows, keep in mind that the bitwise operators are applied to each individual bit within each operand.

Tabel 1.3

A	B	A | B	A & B	A ^ B	~A
0	0	0	0	0	1
1	0	1	0	1	0
0	1	1	0	1	1
1	1	1	1	0	0

The Bitwise NOT:

Also called the bitwise complement, the unary NOT operator, ~, inverts all of the bits of its operand. For example, the number 42, which has the following bit pattern:

00101010 becomes 11010101after the NOT operator is applied.

The Bitwise AND :

The AND operator, &, produces a 1 bit if both operands are also 1. A zero is produced in all other cases. Here is an example:

```
00101010     42
&00001111   15
——————————
00001010     10
```

The Bitwise OR :

The OR operator, |, combines bits such that if either of the bits in the operands is a 1, then the resultant bit is a 1, as shown here:

```
00101010       42
| 00001111     15
——————————
00101111       47
```

The Bitwise XOR:

The XOR operator, ^, combines bits such that if exactly one operand is 1, then the result is 1. Otherwise, the result is zero. The following example shows the effect of the ^. This example also demonstrates a useful attribute of the XOR operation. Notice how the bit pattern of 42 is inverted wherever the second operand has a 1 bit. Wherever the second operand has a 0 bit, the first operand is unchanged. You will find this property useful when performing some types of bit manipulations.

```
00101010      42
^00001111    15
—————————-
00100101      37
```

Using the Bitwise Logical Operators

The following program demonstrates the bitwise logical operators:

```
// Demonstrate the bitwise logical operators.
classBitLogic {
public static void main(String args[]) { String binary[] = {
"0000", "0001", "0010", "0011", "0100", "0101", "0110",
"0111",
"1000", "1001", "1010", "1011", "1100", "1101", "1110", "1111"
};
int a = 3; // 0 + 2 + 1 or 0011 in binary
int b = 6; // 4 + 2 + 0 or 0110 in binary int c = a | b;
int d = a & b;
int e = a ^ b;
int f = (~a & b) | (a & ~b);
int g = ~a & 0x0f;

System.out.println("        a = " + binary[a]);
System.out.println("        b = " + binary[b]);
System.out.println("      a|b = " + binary[c]);
System.out.println("      a&b = " + binary[d]);
 System.out.println("      a^b = " + binary[e]);
 System.out.println("~a&b|a&~b = " + binary[f]);
 System.out.println("       ~a = " + binary[g]);
}
}
```

In this example, **a** and **b** have bit patterns which present all four possibilities for two binary digits: 0-0, 0-1, 1-0, and 1-1. You can see how the | and & operate on each bit by the results in c and d. The values assigned to e and f are the same and illustrate how the^ works. The string array named binary holds the human-readable, binary representation of the numbers 0 through 15. In this example, the array is indexed to show the binary representation of each result. The array is constructed such that the correct string representation of a binary value n is stored in binary[n]. The value of ~a is ANDed with- 64 - 0x0f (0000 1111 in binary) in order to reduce its value to less than 16, so it can be printed by use of the binary array. Here is the output from this program:

```
a = 0011 b = 0110 a|b = 0111 a&b = 0010 a^b = 0101
~a&b|a&~b = 0101
~a = 1100
```

The Left Shift :

The left shift operator, <<, shifts all of the bits in a value to the left a specified number of times. It has this general form:

value<<num

Here, num specifies the number of positions to left-shift the value in value. That is, the << moves all of the bits in the specified value to the left by the number of bit positions specified by num. For each shift left, the high-order bit is shifted out (and lost), and a zero is brought in on the right. This means that when a left shift is applied to an int operand, bits are lost once they are shifted past bit position 31. If the operand is a long, then bits are lost after bit position 63.

Java's automatic type promotions produce unexpected results when you are shifting byte and short values. As you know, byte and short values are promoted to int when an expression is evaluated. Furthermore, the result of such an expression is also an int.

This means that the outcome of a left shift on a byte or short value will be an int, and the bits shifted left will not be lost until they shift past bit position 31. Furthermore, a negative byte or short value will be sign-extended when it is promoted to int. Thus, the high-order bits will be filled with 1's. For these reasons, to perform a left shift on a byte or short implies that you must discard the high-order bytes of the int result. For example, if you

left-shift a byte value, that value will first be promoted to int and then shifted. This means that you must discard the top three bytes of the result if what you want is the result of a shifted byte value. The easiest way to do this is to simply cast the result back into a byte. The following program demonstrates this concept:

```
// Left shifting a byte value. classByteShift {
public static void main(String args[]) {
byte a = 64, b;
int i;
i = a << 2;
b = (byte) (a << 2);
System.out.println("Original value of a: " + a);
 System.out.println("i and b: " + i + " " + b);
}
}
```

The output generated by this program is shown here:

```
Original value of a: 64
 i and b: 256 0
```

Since a is promoted to int for the purposes of evaluation, left-shifting the value 64 (01000000) twice results in i containing the value 256 (1 0000 0000). However, the value in bcontains 0 because after the shift, the low-order byte is now zero. Its only 1 bit has beenshifted out.

Since each left shift has the effect of doubling the original value, programmers frequently use this fact as an efficient alternative to multiplying by 2. But you need to watch out. If you shift a 1 bit into the high-order position (bit 31 or 63), the value will become negative. The following program illustrates this point:

```
// Left shifting as a quick way to multiply by 2.
classMultByTwo {
public static void main(String args[])
{
int i;
intnum = 0xFFFFFFE;
for(i=0; i<4; i++)
{        num = num<< 1;
System.out.println(num);
}
}
}
```

The program generates the following output:

```
536870908
1073741816
2147483632
-32
```

The starting value was carefully chosen so that after being shifted left 4 bit positions, it would produce -32. As you can see, when a 1 bit is shifted into bit 31, the number is interpreted as negative.

The Right Shift:

The right shift operator, >>, shifts all of the bits in a value to the right a specified number of times. Its general form is shown here:

```
value>>num
```

Here, num specifies the number of positions to right-shift the value in value. That is, the>>moves all of the bits in the specified value to the right the number of bit positions specified by num.

The following code fragment shifts the value 32 to the right by two positions, resulting in abeing set to 8:

```
int a = 32;
a = a >> 2; // a now contains 8
```

When a value has bits that are "shifted off," those bits are lost. For example, the next code fragment shifts the value 35 to the right two positions, which causes the two low- order bits to be lost, resulting again in a being set to 8.

```
int a = 35;
a = a >> 2; // a still contains 8
```

Looking at the same operation in binary shows more clearly how this happens:

```
00100011      35
>> 2

00001000       8
```

Each time you shift a value to the right, it divides that value by two-and discards any remainder. You can take advantage of this for high-performance integer division by 2. Of course, you must be sure that you are not shifting any bits off the right end.

When you are shifting right, the top (leftmost) bits exposed by the right shift are filled in with the previous contents of the top bit. This is called sign extension and serves to preserve the sign of negative numbers when you shift them right.

For example, -8 >> 1 is-4, which, in binary, is11111000
 -8>>111111100   -4

It is interesting to note that if you shift -1 right, the result always remains -1, since sign extension keeps bringing in more ones in the high-order bits.

Sometimes it is not desirable to sign-extend values when you are shifting them to the right. For example, the following program converts a byte value to its hexadecimal string representation. Notice that the shifted value is masked by ANDing it with 0x0f to discard any sign-extended bits so that the value can be used as an index into the array of hexadecimal characters.

```
// Masking sign extension.
classHexByte {
static public void main(String args[]) {
char hex[] = {
'0', '1', '2', '3', '4', '5', '6', '7',
'8', '9', 'a', 'b', 'c', 'd', 'e', 'f'
};
byte b = (byte) 0xf1;
System.out.println("b = 0x" + hex[(b >> 4) & 0x0f] + hex[b &0x0f]);
}
}
```

Here is the output of this program:

```
b = 0xf1
```

The Unsigned Right Shift :

As you have just seen, the >> operator automatically fills the high-order bit with its previous contents each time a shift occurs. This preserves the sign of the value. However, sometimes this is undesirable. For example, if you are shifting something that does not represent a numeric value, you may not want sign extension to take place. Thissituation is common when you are working with pixel-based values and graphics. In these cases you will generally want to shift a zero into the high-order bit no matter what itsinitial value was. This is known as an unsigned shift. To accomplish this, you will useJava's unsigned, shift-right operator, >>>, which always shifts zeros into the high-order bit.

The following code fragment demonstrates the >>>. Here, a is set to -1, which sets all 32 bits to 1 in binary. This value is then shifted right 24 bits, filling the top 24 bits with zeros, ignoring normal sign extension. This sets a to 255.

```
int a = -1;
a = a >>> 24;
```

Here is the same operation in binary form to further illustrate what is happening:

```
11111111 11111111 11111111 11111111     -1  in binary as an int
>>>24
00000000 00000000 00000000 11111111   255  in binary as an int
```

The >>> operator is often not as useful as you might like, since it is only meaningful for32- and 64-bit values. Remember, smaller values are automatically promoted to int in expressions. This means that sign-extension occurs and that the shift will take place on a32-bit rather than on an 8- or 16-bit value. That is, one might expect an unsigned right shift on a byte value to zero-fill beginning at bit 7. But this is not the case, since it is a 32-bit value that is actually being shifted. The following program demonstrates this effect:

```
// Unsigned shifting a byte value.
classByteUShift {
static public void main(String args[]) {
char hex[] = {
'0', '1', '2', '3', '4', '5', '6', '7',
'8', '9', 'a', 'b', 'c', 'd', 'e', 'f'
};
byte b = (byte) 0xf1;
byte c = (byte) (b >> 4);
byte d = (byte) (b >>> 4);
byte e = (byte) ((b & 0xff) >> 4);

System.out.println("                b = 0x"+ hex[(b >> 4) & 0x0f] +
hex[b & 0x0f]);
System.out.println("         b >> 4 = 0x"+ hex[(c >> 4) & 0x0f] +
hex[c & 0x0f]); System.out.println("        b >>> 4 = 0x"+ hex[(d
>> 4) & 0x0f] + hex[d & 0x0f]); System.out.println("(b & 0xff) >>
4 = 0x"+ hex[(e >> 4) & 0x0f] + hex[e & 0x0f]);
}
}
```

The following output of this program shows how the >>> operator appears to do nothing when dealing with bytes. The variable b is set to an arbitrary negative byte value for this demonstration. Then c is assigned the byte value of b shifted right by four, which is 0xff because of the expected sign extension. Then d is assigned the byte value of b unsigned shifted right by four, which you might have expected to be 0x0f, but is actually 0xff because of the sign extension that happened when b was promoted to int before theshift. The last expression sets e to the byte value of b masked to 8 bits using the AND operator, then shifted right by four, which produces the expected value of 0x0f. Notice that the unsigned shift right operator was not used for d, since the state of the sign bitafter the AND was known.

```
b = 0xf1 b >> 4 = 0xff
b>>> 4 = 0xff
(b& 0xff) >> 4 = 0x0f
```

Bitwise Operator Assignments :

All of the binary bitwise operators have a shorthand form similar to that of the algebraic operators, which combines the assignment with the bitwise operation. For example, the following two statements, which shift the value in a right by four bits, are equivalent:

```
a = a >> 4;
a >>= 4;
```

Likewise, the following two statements, which result in a being assigned the bitwise expression a OR b, are equivalent:

```
a = a | b;
a |= b;
```

The following program creates a few integer variables and then uses the shorthand form of bitwise operator assignments to manipulate the variables:

```
classOpBitEquals {
public static void main(String args[]) {
int a = 1; int b = 2; int c = 3;

a |= 4;
b >>= 1; c <<= 1; a ^= c;
System.out.println("a = " + a); System.out.println("b = " + b);
System.out.println("c = " + c);
}
}
```

The output of this program is shown here:

```
a = 3 b = 1 c = 6
```

Relational Operators:

The relational operators determine the relationship that one operand has to the other. Specifically, they determine equality and ordering. The relational operators are shown here:

Operator	**Result**
==	Equal to
!=	Not equal to
>	Greater than
<	Less than
>=	Greater than or equal to
<=	Less than or equal to

The outcome of these operations is a boolean value. The relational operators are most frequently used in the expressions that control the if statement and the various loop statements.

Any type in Java, including integers, floating-point numbers, characters, and Booleans can be compared using the equality test, ==, and the inequality test, !=. Notice that in

Java (as in C and C++) equality is denoted with two equal signs, not one. (Remember: a single equal sign is the assignment operator.) Only numeric types can be compared using the ordering operators. That is, only integer, floating-point, and character operands may be compared to see which is greater or less than the other.

As stated, the result produced by a relational operator is a boolean value. For example, the following code fragment is perfectly valid:

```
int a = 4;
int b = 1;
boolean c = a < b;
```

In this case, the result of a<b (which is false) is stored in c.

If you are coming from a C/C++ background, please note the following. In C/C++, these types of statements are very common:

```
int done;
// ...
if(!done) ... // Valid in C/C++
if(done) ...   // but not in Java.
```

In Java, these statements must be written like this:

```
        if(done == 0)) ... // This is Java-style. if(done != 0) ...
```

The reason is that Java does not define true and false in the same way as C/C++. In C/C++, true is any nonzero value and false is zero. In Java, true and false are nonnumeric values which do not relate to zero or nonzero. Therefore, to test for zero or nonzero, you must explicitly employ one or more of the relational operators.

Boolean Logical Operators :

The Boolean logical operators shown here operate only on boolean operands. All of the binary logical operators combine two boolean values to form a resultant boolean value.

Tabel 1.4

Operator	Result
&	Logical AND
|	Logical OR
^	Logical XOR (exclusive OR)
||	Short-circuit OR
&&	Short-circuit AND
!	Logical unary NOT
&=	AND assignment
|=	OR assignment
^=	XOR assignment
==	Equal to
!=	Not equal to
?:	Ternary if-then-else

The logical Boolean operators, &, |, and ^, operate on boolean values in the same way that they operate on the bits of an integer. The logical !operator inverts the Boolean state: !true

== false and !false == true. The following table shows the effect of each logical operation:

A	**B**	**A | B**	**A & B**	**A ^ B**	**!A**
False	False	False	True		
True	False	True	False	True	False
False	True	True	False	True	True
True	True	False	False		

Here is a program that is almost the same as the BitLogic example shown earlier, but it operates on boolean logical values instead of binary bits:

```
// Demonstrate the boolean logical operators.
classBoolLogic {
public static void main(String args[]) {
boolean a = true; boolean b = false; boolean c = a | b; boolean d = a & b; boolean e = a ^ b;
boolean f = (!a & b) | (a & !b);
boolean g = !a;
System.out.println("      a = " + a);
System.out.println("      b = " + b); System.out.println("     a|b = " + c); System.out.println("     a&b = " + d); System.out.println("     a^b = " + e); System.out.println("!a&b|a&!b = " + f); System.out.println("      !a = " + g);
}
}
```

After running this program, you will see that the same logical rules apply to boolean values as they did to bits. As you can see from the following output, the string representation of a Java boolean value is one of the literal values true or false:

```
a = true
b = false
a|b = true a&b = false a^b = true
a&b|a&!b = true
!a = false
```

Short-Circuit Logical Operators :

Java provides two interesting Boolean operators not found in most other computer languages. These are secondary versions of the Boolean AND and OR operators, and are known as short-circuit logical operators. As you can see from the preceding table, the OR operator results in true when A is true, no matter what B is. Similarly, the AND operator results in false when A is false, no matter what B is. If you use the || and && forms, rather than the | and & forms of these operators, Java will not bother to evaluate the right-hand operand when the outcome of the expression can be determined by theleft operand alone. This is very useful when the right-hand operand depends on the left one being true or false in order to function properly. For example, the following code fragment shows how you can take advantage of short-circuit logical evaluation to be sure that a division operation will be valid before evaluating it:

```
if (denom != 0 &&num / denom> 10)
```

Since the short-circuit form of AND (&&) is used, there is no risk of causing a run-time exception when denom is zero. If this line of code were written using the single & version of AND, both sides would have to be evaluated, causing a run-time exception when denom is zero.

It is standard practice to use the short-circuit forms of AND and OR in cases involving Boolean logic, leaving the single-character versions exclusively for bitwise operations. However, there are exceptions to this rule.]

For example, consider the following statement:

```
if(c==1 & e++ < 100) d = 100;
```

Here, using a single & ensures that the increment operation will be applied to e whether c is equal to 1 or not.

The Assignment Operator :

You have been using the assignment operator since Chapter 2. Now it is time to take a formal look at it. The assignment operator is the single equal sign, =. The assignment operator works in Java much as it does in any other computer language. It has this general form:

```
var = expression;
```

Here, the type of var must be compatible with the type of expression.

The assignment operator does have one interesting attribute that you may not be familiar with: it allows you to create a chain of assignments. For example, consider this fragment:

```
int x, y, z;

x = y = z = 100; // set x, y, and z to 100
```

This fragment sets the variables x, y, and z to 100 using a single statement. This works because the = is an operator that yields the value of the right-hand expression. Thus, the value of z = 100 is 100, which is then assigned to y, which in turn is assigned to x. Using a "chain of assignment" is an easy way to set a group of variables to a common value.

The ? Operator:

Java includes a special ternary (three-way) operator that can replace certain types of if- then-else statements. This operator is the ?, and it works in Java much like it does in C and C++. It can seem somewhat confusing at first, but the ?can be used very effectively

once mastered. The ?has this general form:
expression1 ?expression2 : expression3

Here, expression1 can be any expression that evaluates to a boolean value. If expression1 is true, then expression2 is evaluated; otherwise, expression3 is evaluated. The result of the ?operation is that of the expression evaluated. Both expression2 and expression3 are required to return the same type, which can't be void.

Here is an example of the way that the ?is employed:

```
ratio = denom == 0 ? 0 :num / denom;
```

When Java evaluates this assignment expression, it first looks at the expression to the left of the question mark. If denom equals zero, then the expression between the question mark and the colon is evaluated and used as the value of the entire ?expression. If denom does not equal zero, then the expression after the colon isevaluated and used for the value of the entire ? expression. The result produced by the ?operator is then assigned to ratio.

Here is a program that demonstrates the ?operator. It uses it to obtain the absolute value of a variable.

```
// Demonstrate ?.
class Ternary {
public static void main(String args[]) {
int i, k;
i = 10;
k = i <0 ? -i : i; // get absolute value of i
System.out.print("Absolute value of "); System.out.println(i + " is
" + k);

i = -10;
k = i <0 ? -i : i; // get absolute value of i
System.out.print("Absolute value of "); System.out.println(i + " is
" + k);
}
}
```

output

```
Absolute value of 10 is 10
Absolute value of -10 is 10
```

Operator Precedence :

Following table shows the order of precedence for Java operators, from highest to lowest. Notice that the first row shows items that you may not normally think of as operators: parentheses, square brackets, and the dot operator. Parentheses are used to alter the precedence of an operation. As you know from the previous chapter, the square brackets provide array indexing. The dot operator is used to dereference objects and will be discussed later in this book.

Table 1-5.

The Precedence of the Java Operators:

Highest

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

Lowest

Using Parentheses :

Parentheses raise the precedence of the operations that are inside them. This is often necessary to obtain the result you desire. For example, consider the following expression:

```
a >> b + 3
```

This expression first adds 3 to b and then shifts a right by that result. That is, this expression can be rewritten using redundant parentheses like this:

```
a >> (b + 3)
```

However, if you want to first shift a right by b positions and then add 3 to that result, you will need to parenthesize the expression like this:
(a >> b) + 3

In addition to altering the normal precedence of an operator, parentheses can sometimes be used to help clarify the meaning of an expression. For anyone reading your code, a complicated expression can be difficult to understand. Adding redundant but clarifying parentheses to complex expressions can help prevent confusion later. For example,

which of the following expressions is easier to read?
a | 4 + c >> b & 7
(a | (((4 + c) >> b) & 7))

One other point: parentheses (redundant or not) do not degrade the performance of your program. Therefore, adding parentheses to reduce ambiguity does not negatively affect your program.

1.5 BRANCHING AND LOOPING BRANCHING

Overview :

A programming language uses *control* statements to cause the flow of execution toadvance and branch based on changes to the state of a program. Java's program controlstatements can be put into the following categories: selection, iteration, and jump.*Selection* statements allow your program to choose different paths of execution basedupon the outcome of an expression or the state of a variable. *Iteration* statements enableprogram execution to repeat one or more statements (that is, iteration statements form loops). *Jump* statements allow your program to execute in a nonlinear fashion. All ofJava's control statements are examined here.

Java's Selection Statements :

Java supports two selection statements: if and switch. These statements allow you tocontrol the flow of your program's execution based upon conditions known only duringrun time. If your background inprogramming does not include C/C++, you will bepleasantly surprised by the power and flexibility contained in these two statements.

If Statement:

The ifstatement is Java's conditional branch statement. It can be used to route programexecution through two different paths. Here is the general form of the if statement:
if (*condition*) *statement1*; else *statement2*;

Here, each *statement* may be a single statement or a compound statement enclosed incurly braces (that is, a *block*). The *condition* is any expression that returns a boolean value. The else clause is optional.The if works like this: If the *condition* is true, then *statement1* is executed. Otherwise,*statement2* (if it exists) is executed. In no case will both statements be executed. Forexample, consider the following:

```
int a, b;
// ...
if(a < b) a = 0;
else b = 0;
```

Here, if a is less than b, then a is set to zero. Otherwise, b is set to zero. In no case arethey both set to zero.Most often, the expression used to control the if will involve the relational operators.However, this is not technically necessary. It is possible to control the if using a singleboolean variable, as shown in this code fragment:

```
boolean dataAvailable;
// ...
if (dataAvailable)
ProcessData();
else
waitForMoreData();
```

Remember, only one statement can appear directly after the if or the else. If you want toinclude more statements, you'll need to create a block, as in this fragment:

```
int bytesAvailable;
// ...
if (bytesAvailable > 0) {
ProcessData();
bytesAvailable -= n;
} else
waitForMoreData();
```

Here, both statements within the if block will execute if bytesAvailable is greater thanzero.Some programmers find it convenient to include the curly braces when using the if, evenwhen there is only onestatement in each clause. This makes it easy to add anotherstatement at a later date, and you don't have to worry about forgetting the braces. In fact, forgetting to define a block when one is needed is a common cause of errors. For example, consider the following code fragment:

```
int bytesAvailable;
// ...
if (bytesAvailable > 0) {
ProcessData();
bytesAvailable -= n;
} else
waitForMoreData();
bytesAvailable = n;
```

It seems clear that the statement bytesAvailable = n; was intended to be executedinside the else clause, because of the indentation level. However, as you recall,whitespace is insignificant to Java, and there is no way for the compiler to know what wasintended. This code will compile without complaint, but it will behave incorrectly when run.

The preceding example is fixed in the code that follows:

```
int bytesAvailable;
// ...
if (bytesAvailable > 0) {
ProcessData();
bytesAvailable -= n;
} else {
waitForMoreData();
bytesAvailable = n;
}
```

Nested ifs :

A *nested* if is an if statement that is the target of another if or else. Nested ifs are verycommon in programming. When you nest ifs, the main thing to remember is that an elsestatement always refers to the nearest if statement that is within the same block as them else and that is not already associated with an else. Here is an example:

```
if(i == 10) {
if(j < 20) a = b;
if(k > 100) c = d; // this if is
else a = c; // associated with this else
}
else a = d; // this else refers to if(i == 10)
```

As the comments indicate, the final else is not associated with if(j<20), because it is notin the same block (even though it is the nearest if without an else). Rather, the final elseis associated with if(i==10). The inner else refers to if(k>100), because it is the closest ifwithin the same block.

The if-else-if Ladder:

A common programming construct that is based upon a sequence of nested ifs is the *ifelse-*

if ladder. It looks like this:

```
if(condition)
statement;
else if(condition)
statement;
else if(condition)
statement;
.
.
.
else
statement;
```

The if statements are executed from the top down. As soon as one of the conditionscontrolling the if is true, the statement associated with that if is executed, and the rest ofthe ladder is bypassed. If none of the conditions is true, then the final else statement willbe executed. The final else acts as a default condition; that is, if all other conditional testsfail, then the last else statement is performed. If there is no final else and all otherconditions are false, then no action will take place.
Here is a program that uses an if-else-if ladder to determine which season a particularmonth is in.

```
// Demonstrate if-else-if statements.
class IfElse {
public static void main(String args[]) {
int month = 4; // April
String season;
if(month == 12 || month == 1 || month == 2)
season = "Winter";
else if(month == 3 || month == 4 || month == 5)
season = "Spring";
else if(month == 6 || month == 7 || month == 8)
season = "Summer";
else if(month == 9 || month == 10 || month == 11)
season = "Autumn";
else
season = "Bogus Month";
System.out.println("April is in the " + season + ".");
}
}
```

Here is the output produced by the program:

```
April is in the Spring.
```

You might want to experiment with this program before moving on. As you will find, no

matter what value you give month, one and only one assignment statement within the

ladder will be executed.

Switch:

The switch statement is Java's multiway branch statement. It provides an easy way todispatch execution to different parts of your code based on the value of an expression.As such, it often provides a better alternative than a large series of if-else-if statements.

Here is the general form of a switch statement:

```
switch (expression) {
case value1:
// statement sequence
break;
case value2:
// statement sequence
break;
.
.
.
case valueN:
// statement sequence
break;
default:
// default statement sequence
}
```

The *expression* must be of type byte, short, int, or char; each of the *values* specified inthe case statements must be of a type compatible with the expression. Each case valuemust be a unique literal (that is, it must be a constant, not a variable). Duplicate casevalues are not allowed.

The switch statement works like this: The value of the expression is compared with eachof the literal values in the case statements. If a match is found, the code sequencefollowing that case statement is executed. If none of the constants matches the value ofthe expression, then the default statement is executed. However, the default statementis optional. If no case matches and no default is present, then no further action is taken.The break statement is used inside the switch to terminate a statement sequence. Whena break statement is encountered, execution branches to the first line of code that followsthe entire switch statement. This has the effect of "jumping out" of the switch.

Here is a simple example that uses a switch statement:

```
// A simple example of the switch.
class SampleSwitch {
public static void main(String args[]) {
for(int i=0; i<6; i++)
switch(i) {
case 0:
System.out.println("i is zero.");
break;
case 1:
System.out.println("i is one.");
break;
case 2:
System.out.println("i is two.");
break;
case 3:
System.out.println("i is three.");
break;
default:
System.out.println("i is greater than 3.");
}
}
}
```

The output produced by this program is shown here:

```
i is zero.
i is one.
i is two.
i is three.
i is greater than 3.
i is greater than 3.
```

As you can see, each time through the loop, the statements associated with the caseconstant that matches i are executed. All others are bypassed. After i is greater than 3,no case statements match, so the default statement is executed.The break statement is optional. If you omit the break, execution will continue on into thenext case. It is sometimes desirable to have multiple cases without break statementsbetween them. For example, consider the following program:

```
// In a switch, break statements are optional.
class MissingBreak {
public static void main(String args[]) {
for(int i=0; i<12; i++)
switch(i) {
case 0:
case 1:
case 2:
case 3:
case 4:
System.out.println("i is less than 5");
break;
case 5:
case 6:
case 7:
case 8:
case 9:
System.out.println("i is less than 10");
break;
default:
System.out.println("i is 10 or more");
}
}
}
```

This program generates the following output:

```
i is less than 5
i is less than 5
i is less than 5
i is less than 5
i is less than 5
i is less than 10
i is less than 10
i is less than 10
i is less than 10
i is less than 10
i is 10 or more
i is 10 or more
```

As you can see, execution falls through each case until a break statement (or the end ofthe switch) is reached.While the preceding example is, of course, contrived for the sake of illustration, omittingthe break statement has many practical applications in real programs. To sample itsmore realistic usage, consider the following rewrite of the season example shown earlier.This version uses a switch to provide a more efficient implementation.

```
// An improved version of the season program.
class Switch {
public static void main(String args[]) {
int month = 4;
String season;
switch (month) {
case 12:
case 1:
case 2:
season = "Winter";
break;
case 3:
```

```
case 4:
case 5:
season = "Spring";
break;
case 6:
case 7:
case 8:
season = "Summer";
break;
case 9:
case 10:
case 11:
season = "Autumn";
break;
default:
season = "Bogus Month";
}
System.out.println("April is in the " + season + ".");
}
}
```

Nested switch Statements :

You can use a switch as part of the statement sequence of an outer switch. This iscalled a *nested* switch. Since a switch statement defines its own block, no conflicts arisebetween the case constants in the inner switch and those in the outer switch. Forexample, the following fragment is perfectly valid:

```
switch(count) {
case 1:
switch(target) { // nested switch
case 0:
System.out.println("target is zero");
break;
case 1: // no conflicts with outer switch
System.out.println("target is one");
break;
}
break;
case 2: // ...
```

Here, the case 1: statement in the inner switch does not conflict with the case 1: statement in the outer switch. The count variable is only compared with the list of casesat the outer level. If count is 1, then target is compared with the inner list cases.In summary, there are three important features of the switch statement to note:

- The switch differs from the if in that switch can only test for equality, whereas if canevaluate any type of Boolean expression. That is, the switch looks only for a matchbetween the value of the expression and one of its case constants.

- No two case constants in the same switch can have identical values. Of course, aswitch statement enclosed by an outer switch can have case constants in common.

- A switch statement is usually more efficient than a set of nested ifs.

The last point is particularly interesting because it gives insight into how the Java compilerworks. When it compiles a switch statement, the Java compiler will inspect each of thecase constants and create a "jump table" that it will use for selecting the path of executiondepending on the value of the expression. Therefore, if you need to select among a largegroup of values, a switch statement will run much faster than the equivalent logic codedusing a sequence of if-elses. The compiler can do this because it knows that the caseconstants are all the same type and simply must be compared for equality with the switchexpression. The compiler has no such knowledge of a long list of if expressions.

Looping :

Iteration Statements:

Java's iteration statements are for, while, and do-while. These statements create whatwe commonly call *loops.* As you probably know, a loop repeatedly executes the same setof instructions until a termination condition is met. As you will see, Java has a loop to fitany programming need.

While :

The while loop is Java's most fundamental looping statement. It repeats a statement orblock while its controlling expression is true. Here is its general form:

```
while(condition) {
// body of loop
}
```

The *condition* can be any Boolean expression. The body of the loop will be executed aslong as the conditional expression is true. When *condition* becomes false, control passesto the next

line of code immediately following the loop. The curly braces are unnecessaryif only a single statement is being repeated.
Here is a while loop that counts down from 10, printing exactly ten lines of "tick":

```
// Demonstrate the while loop.
class While {
public static void main(String args[]) {
int n = 10;
while(n > 0) {
System.out.println("tick " + n);
n—;
}
}
}
```

When you run this program, it will "tick" ten times:

```
tick 10
tick 9
tick 8
tick 7
tick 6
tick 5
tick 4
tick 3
tick 2
tick 1
```

Since the while loop evaluates its conditional expression at the top of the loop, the bodyof the loop will not execute even once if the condition is false to begin with. For example,in the following fragment, the call to println( ) is never executed:

```
int a = 10, b = 20;
while(a > b)
System.out.println("This will not be displayed");
```

The body of the while (or any other of Java's loops) can be empty. This is because a *nullstatement* (one that consists only of a semicolon) is syntactically valid in Java. Forexample, consider the following program:

```
// The target of a loop can be empty.
class NoBody {
public static void main(String args[]) {
int i, j;
i = 100;
j = 200;
// find midpoint between i and j
while(++i < —j) ; // no body in this loop
System.out.println("Midpoint is " + i);
}
}
```

This program finds the midpoint between i and j. It generates the following output:

```
Midpoint is 150
```

Here is how the while loop works. The value of i is incremented, and the value of j isdecremented. These values are then compared with one another. If the new value of i isstill less than the new value of j, then the loop repeats. If i is equal to or greater than j,the loop stops. Upon exit from the loop, i will hold a value that is midway between theoriginal values of i and j. (Of course, this procedure only works when i is less than j tobegin with.) As you can see, there is no need for a loop body; all of the action occurswithin the conditional expression, itself. In professionally written Java code, short loopsare frequently coded without bodies when the controlling expression can handle all of the details itself.

do-while :

As you just saw, if the conditional expression controlling a while loop is initially false,then the body of the loop will not be executed at all. However, sometimes it is desirable toexecute the body of a while loop at least once, even if the conditional expression is falseto begin with. In other words, there are times when you would like to test the terminationexpression at the end of the loop rather than at the beginning. Fortunately, Java suppliesa loop that does just that: the do-while. The do-while loop always executes its body atleast once, because its conditional expression is at the bottom of the loop. Its generalform is:

```
do {
// body of loop
} while (condition);
```

Each iteration of the do-while loop first executes the body of the loop and then evaluatesthe conditional expression. If this expression is true, the loop will repeat. Otherwise, theloop terminates. As with all of Java's loops, *condition* must be a Boolean expression.

Here is a reworked version of the "tick" program that demonstrates the do-while loop. Itgenerates the same output as before.

```
// Demonstrate the do-while loop.
class DoWhile {
public static void main(String args[]) {
int n = 10;
do {
System.out.println("tick " + n);
n—;
} while(n > 0);
}
}
```

The loop in the preceding program, while technically correct, can be written moreefficiently as follows:

```
do {
System.out.println("tick " + n);
} while(—n > 0);
```

In this example, the expression (– –n > 0) combines the decrement of n and the test forzero into one expression. Here is how it works. First, the – –n statement executes,decrementing n and returning the new value of n. This value is then compared with zero.If it is greater than zero, the loop continues; otherwise it terminates.

The do-while loop is especially useful when you process a menu selection, because youwill usually want the body of a menu loop to execute at least once. Consider the following

program which implements a very simple help system for Java's selection and iterationstatements:

```
// Using a do-while to process a menu selection
class Menu {
public static void main(String args[])
throws java.io.IOException {
char choice;
do {
System.out.println("Help on:");
System.out.println(" 1. if");
System.out.println(" 2. switch");
System.out.println(" 3. while");
System.out.println(" 4. do-while");
System.out.println(" 5. for\\n");
System.out.println("Choose one:");
choice = (char) System.in.read();
} while( choice < '1' || choice > '5');
System.out.println("\\n");
switch(choice) {
case '1':
System.out.println("The if:\\n");
System.out.println("if(condition) statement;");
System.out.println("else statement;");
break;
case '2':
System.out.println("The switch:\\n");
System.out.println("switch(expression) {");
System.out.println(" case constant:");
System.out.println(" statement sequence");
System.out.println(" break;");
System.out.println(" // ...");
System.out.println("}");
break;
case '3':
System.out.println("The while:\\n");
System.out.println("while(condition) statement;");
break;
case '4':
System.out.println("The do-while:\\n");
System.out.println("do {");
System.out.println(" statement;");
System.out.println("} while (condition);");
break;
case '5':
System.out.println("The for:\\n");
System.out.print("for(init; condition; iteration)");
System.out.println(" statement;");
break;
}
}
}
```

Here is a sample run produced by this program:

```
Help on:
1. if
2. switch
3. while
4. do-while
5. for
Choose one:
4
The do-while:
```

```
do {
statement;
} while (condition);
```

In the program, the do-while loop is used to verify that the user has entered a validchoice. If not, then the user is reprompted. Since the menu must be displayed at leastonce, the do-while is the perfect loop to accomplish this.A few other points about this example: Notice that characters are read from the keyboardby calling System.in.read( ). This is one of Java's console input functions. System.in.read( ) is used here to obtain the user's choice. It reads characters fromstandard input (returned as integers, which is why the return value was cast to char). Bydefault, standard input is line buffered, so you must press ENTER before any charactersthat you type will be sent to your program.Java's console input is quite limited and awkward to work with. Further, most real-worldJava programs and applets will be graphical and window-based. However, it is useful in thiscontext. One other point: Because System.in.read( ) is being used, the program mustspecify the throws java.io.IOException clause. This line is necessary to handle inputerrors. It is part of Java's exception handling features.

ForLoop :

You were introduced to a simple form of the for loop. As you will see, it is apowerful and versatile construct. Here is the general form of the for statement:

```
for(initialization; condition; iteration) {
// body
}
```

If only one statement is being repeated, there is no need for the curly braces.The for loop operates as follows. When the loop first starts, the *initialization* portion of theloop is executed. Generally, this is an expression that sets the value of the *loop controlvariable,* which acts as a counter that controls the loop. It is important to understand thatthe initialization expression is only executed once. Next, *condition* is evaluated. This mustbe a Boolean expression. It usually tests the loop control variable against a target value.If this expression is true, then the body of the loop is executed. If it is false, the loopterminates. Next, the *iteration* portion of the loop is executed. This is usually anexpression that increments or decrements the loop control variable. The loop theniterates, first evaluating the conditional expression, then executing the body of the loop,and then executing the iteration expression with each pass. This process repeats until thecontrolling expression is false.

Here is a version of the "tick" program that uses a for loop:

```
// Demonstrate the for loop.
class ForTick {
public static void main(String args[]) {
int n;
for(n=10; n>0; n—)
System.out.println("tick " + n);
}
}
```

Declaring Loop Control Variables Inside the for Loop :

Often the variable that controls a for loop is only needed for the purposes of the loop andis not used elsewhere. When this is the case, it is possible to declare the variable insidethe initialization portion of the for. For example, here is the preceding program recodedso that the loop control variable n is declared as an int inside the for:

```
// Declare a loop control variable inside the for.
class ForTick {
public static void main(String args[]) {
// here, n is declared inside of the for loop
for(int n=10; n>0; n—)
System.out.println("tick " + n);
}}
```

When you declare a variable inside a for loop, there is one important point to remember:the scope of that variable ends when the for statement does. (That is, the scope of thevariable is limited to the for loop.) Outside the for loop, the variable will cease to exist. Ifyou need to use the loop control variable elsewhere in your program, you will not be ableto declare it inside the for loop.When the loop control variable will not be needed elsewhere, most Java programmersdeclare it inside the for. For example, here is a simple program that tests for primenumbers. Notice that the loop control variable, i, is declared inside the for since it is notneeded elsewhere.

```
// Test for primes.
class FindPrime {
public static void main(String args[]) {
int num;
boolean isPrime = true;
num = 14;
for(int i=2; i < num/2; i++) {
if((num % i) == 0) {
isPrime = false;
break;
}
}
if(isPrime) System.out.println("Prime");
else System.out.println("Not Prime");
}
}
```

Using the Comma :

There will be times when you will want to include more than one statement in theinitialization and iteration portions of the for loop. For example, consider the loop in thefollowing program:

```
class Sample {
public static void main(String args[]) {
int a, b;
b = 4;
for(a=1; a<b; a++) {
System.out.println("a = " + a);
System.out.println("b = " + b);
b—;
}
}
}
```

As you can see, the loop is controlled by the interaction of two variables. Since the loop isgoverned by two variables, it would be useful if both could be included in the forstatement, itself, instead of b being handled manually. Fortunately, Java provides a wayto accomplish this. To allow two or more variables to control a for loop, Java permits youto include multiple statements in both the initialization and iteration portions of the for.Each statement is separated from the next by a comma.Using the comma, the preceding for loop can be more efficiently coded as shown here:

```
// Using the comma.
class Comma {
public static void main(String args[]) {
int a, b;
for(a=1, b=4; a<b; a++, b—) {
System.out.println("a = " + a);
System.out.println("b = " + b);
}
}
}
```

In this example, the initialization portion sets the values of both a and b. The two commaseparatedstatements in the iteration portion are executed each time the loop repeats.

The program generates the following output:

```
a = 1
b = 4
a = 2
b = 3
```

Some for Loop Variations :

The for loop supports a number of variations that increase its power and applicability.The reason it is so flexible is that its three parts, the initialization, the conditional test, andthe iteration, do not need to be used for only those purposes. In fact, the three sections ofthe for can be used for any purpose you desire. Let's look at some examples.One of the most common variations involves the conditional expression. Specifically, thisexpression does not need to test the loop control variable against some target value. Infact, the condition controlling the for can be any Boolean expression. For example,consider the following fragment:

```
boolean done = false;
for(int i=1; !done; i++) {
// ...
if(interrupted()) done = true;
}
```

In this example, the for loop continues to run until the boolean variable done is set totrue. It does not test the value of i.

Here is another interesting for loop variation. Either the initialization or the iterationexpression or both may be absent, as in this next program:

```
// Parts of the for loop can be empty.
class ForVar {
public static void main(String args[]) {
int i;
boolean done = false;
i = 0;
for( ; !done; ) {
System.out.println("i is " + i);
if(i == 10) done = true;
i++;
}
}
}
```

Here, the initialization and iteration expressions have been moved out of the for. Thus,parts of the for are empty. While this is of no value in this simple example—indeed, itwould be considered quite poor style—there can be times when this type of approachmakes sense. For example, if the initial condition is set through a complex expressionelsewhere in the program or if the loop control variable changes in a nonsequentialmanner determined by actions that occur within the body of the loop, it may beappropriate to leave these parts of the for empty.

Here is one more for loop variation. You can intentionally create an infinite loop (a loopthat never terminates) if you leave all three parts of the for empty. For example:

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

This loop will run forever, because there is no condition under which it will terminate.Although there are some programs, such as operating system command processors, thatrequire an infinite loop, most "infinite loops" are really just loops with special terminationrequirements. As you will soon see, there is a way to terminate a loop—even an infiniteloop like the one shown—that does not make use of the normal loop conditional expression.

Nested Loops :

Like all other programming languages, Java allows loops to be nested. That is, one loopmay be inside another. For example, here is a program that nests for loops:

```
// Loops may be nested.
class Nested {
public static void main(String args[]) {
int i, j;
for(i=0; i<10; i++) {
for(j=i; j<10; j++)
System.out.print(".");
System.out.println();
}
}
}
```

The output produced by this program is shown here:

....

....

....

....

......

.....

1.6 CLASSES , OBJECTS AND METHODS

The class is at the core of Java. It is the logical construct upon which the entire javalanguage is built because it defines the shape and nature of an object. A class is declared by use of the class keyword..

Classes can (andusually do) get much more complex. The general form of a class definition is shownhere:

```
class classname {
type instance-variable1;
type instance-variable2;
// ...
type instance-variableN;
type methodname1(parameter-list) {
// body of method
}
type methodname2(parameter-list) {
// body of method
}
// ...
type methodnameN(parameter-list) {
// body of method
}
}
```

The data, or variables, defined within a class are called *instance variables.* The code iscontained within *methods.* Collectively, the methods and variables defined within a classare called *members* of the class. In most classes, the instance variables are acted uponand accessed by the methods defined for that class. Thus, it is the methods thatdetermine how a class' data can be used.Variables defined within a class are called instance variables because each instance ofthe class (that is, each object of the class) contains its own copy of these variables. Thus,the data for one object is separate and unique from the data for another. We will comeback to this point shortly, but it is an important concept to learn early.All methods have the same general form as main( ), which we have been using thus far.However, most methods will not be specified as static or public. Notice that the generalform of a class does not specify a main( ) method. Java classes do not need to have amain( ) method. You only specify one if that class is the starting point for your programThe data, or variables, defined within a class are called *instance variables.* The code iscontained within *methods.* Variables defined within a class are called instance variables because each instance ofthe class (that is, each object of the class) contains its own copy of these variables.Simple Example: Here is a class called Box thatdefines three instance variables: width, height, and depth.

Here is a program that uses the Box class:

```
/* A program that uses the Box class.
Call this file BoxDemo.java
*/
class Box {
double width;
double height;
double depth;
}
// This class declares an object of type Box.
class BoxDemo {
public static void main(String args[])
{
Box mybox = new Box();
double vol;
// assign values to mybox's instance variables
mybox.width = 10;
mybox.height = 20;
mybox.depth = 15;
// compute volume of box
vol = mybox.width * mybox.height * mybox.depth;
System.out.println("Volume is " + vol);
}
}
```

You should call the file that contains this program BoxDemo.java, because the main( )method is in the class called BoxDemo, not the class called Box.To runthis program, you must execute BoxDemo.class. When you do, you will see thefollowing output:

```
Volume is 3000.0
```

Class Features :

***** A class can have public or default (no modifier) visibility.

* It can be either abstract, final or concrete (no modifier).
* It must have the class keyword, and class must be followed by a legal identifier.
* It may optionally extend one parent class. By default, it will extend java.lang.Object.
* It may optionally implement any number of comma-separated interfaces.
* The class's variables and methods are declared within a set of curly braces '{}'.
* Each .java source file may contain only one public class. A source file may contain any number of default visible classes.
* Finally, the source file name must match the public class name and it must have a .java suffix.

Methods:

This is the general form of a method:

```
type name(parameter-list)
{
// body of method
}
```

Here, *type* specifies the type of data returned by the method. This can be any valid type,including class types that you create. If the method does not return a value, its return typemust be void. The name of the method is specified by *name.*The *parameter-list* is a sequence of type and identifier pairs separated by commas. Parameters are essentially variables that receive the value of the *arguments* passed to the method when it is called. Methods that have a return type other than void return a value to the calling routine using the following form of the return statement:

```
return value;
```

Adding Methods to Class:

```
// This program includes a method inside the box class.
class Box
{
double width;
double height;
double depth;
// display volume of a box
void volume()
{
System.out.print("Volume is ");
System.out.println(width * height * depth);
}
}
class BoxDemo3
{
public static void main(String args[])
{
Box mybox1 = new Box();
Box mybox2 = new Box();
// assign values to mybox1's instance variables
mybox1.width = 10;
mybox1.height = 20;
mybox1.depth = 15;
/* assign different values to mybox2's
instance variables */
mybox2.width = 3;
mybox2.height = 6;
mybox2.depth = 9;
// display volume of first box
mybox1.volume();
// display volume of second box
mybox2.volume();
}
}
```

This program generates the following output, which is the same as the previous version.

```
Volume is 3000.0
Volume is 162.0
```

The following two lines of code:

```
mybox1.volume();
mybox2.volume();
```

The first line here invokes the volume( ) method on mybox1. That is, it calls volume( )relative to the mybox1 object, using the object's name followed by the dot operator.Thus, the call to mybox1.volume( ) displays the volume of the box defined by mybox1,and the call to mybox2.volume( ) displays the volume of the box defined by mybox2.Each time volume( ) is invoked, it displays the volume for the specified box.

Returning a Value :

While the implementation of volume( ) does move the computation of a box's volumeinside the Box class where it belongs, it is not the best way to do it. For example, what ifanother part of your program wanted to know the volume of a box, but not display itsvalue? A better way to implement volume( ) is to have it compute the volume of the boxand return the result to the caller. The following example, an improved version of thepreceding program, does just that:

```
// Now, volume() returns the volume of a box.
class Box
{
double width;
double height;
double depth;
// compute and return volume
double volume()
{
return width * height * depth;
}
}
class BoxDemo4
{
public static void main(String args[])
{
Box mybox1 = new Box();
Box mybox2 = new Box();
double vol;
// assign values to mybox1's instance variables
mybox1.width = 10;
mybox1.height = 20;
mybox1.depth = 15;
/* assign different values to mybox2's
instance variables */
mybox2.width = 3;
mybox2.height = 6;
mybox2.depth = 9;
// get volume of first box
vol = mybox1.volume();
System.out.println("Volume is " + vol);
// get volume of second box
vol = mybox2.volume();
System.out.println("Volume is " + vol);
}
}
```

As you can see, when volume( ) is called, it is put on the right side of an assignmentstatement. On the left is a variable, in this case vol, that will receive the value returned byvolume( ). Thus, aftervol = mybox1.volume();executes, the value of mybox1.volume( ) is 3,000 and this value then is stored in vol.

There are two important things to understand about returning values:

- The type of data returned by a method must be compatible with the return typespecified by the method. For example, if the return type of some method is boolean,you could not return an integer.
- The variable receiving the value returned by a method (such as vol, in this case) mustalso be compatible with the return type specified for the method.

One more point: The preceding program can be written a bit more efficiently becausethere is actually no need for the vol variable. The call to volume( ) could have been usedin the println( ) statement directly, as shown here:

```
System.out.println("Volume is " + mybox1.volume());
```

In this case, when println( ) is executed, mybox1.volume( ) will be called automaticallyand its value will be passed to println( ).

1.7 ARRAYS STRING AND VECTORS

Arrays are generally effective means of storing groups of variables. An array is a group of variables that share the same name and are ordered sequentially from zero to one less than the number of variables in the array. The number of variables that can be stored in an array is called the array's *dimension*. Each variable in the array is called an *element* of the array.
Declaring ArraysLike all other variables in Java an array must be declared. When you declare an array variable you suffix the type with [] to indicate that this ariable is an array. Here are some examples:

```
int[]k;
float[]yt;
String[] names;
```

Allocating Arrays :

To actually create the array (or any other object) use the new operator. When we create an array we need to tell the compiler how many elements will be stored in it. Here's how we'd create the variables declared above:

```
K = new int[3];
yt = new float[7];
names = new String[50];
```

Initializing Arrays :

Individual elements of the array are referenced by the array name and by an integer which represents their position in the array. The numbers we use to identify themare called subscripts or indexes into the array. Subscripts are consecutive integers beginning with 0. Thus the array k above has elements k[0], k[1], and k[2]. Since we started counting at zero there is no k[3], and trying to access it will generate an ArrayIndexOutOfBoundsException. *subscriptsindexes* k k[0] k[1] k[2] k[3] ArrayIndexOutOfBoundsException You can use array elements wherever you'd use a similarly typed variable that wasn't part of an array.

Here's how we'd store values in the arrays we've been working with:

```
k[0]=2;
k[1]=5;
k[2]=-2;
yt[6]=7.5f;
names[4] = "Fred";
```

This step is called initializing the array or, more precisely, initializing the elements of the array.

Two Dimensional Arrays :

Declaring, Allocating and Initializing Two Dimensional ArraysTwo dimensional arrays are declared, allocated and initialized much like one dimensional arrays. However we have to specify two dimensions rather than one, and we typically use two nested for loops to fill the array. for examples above are filled with the sum of their row and column indices. Here's some code that would create and fill such an array:

```
Class FillArray {
    public static void main (String args[]) {
        int[][] M;
        M = new int[4][5];
     for (int row=0; row < 4; row++) {
       for (int col=0; col < 5; col++) {
          M[row][col] = row+col;
       }
     }
    }
 }
```

In two-dimensional arrays ArrayIndexOutOfBounds errors occur whenever you exceed the maximum column index or row index.

1.8 INTERFACES , PACKEGES

*Packages*arecontainers for classes that are used to keep the class name space Compartmentalized. For example, a package allows you to create a class named List,which you can store in your own package without concern that it will collide with someother class named Liststored elsewhere.Using interface, you can specify a set of methods which can be implemented by one or more classes.

Packages :

To create a package is quite easy: simply include a package command as the first
statement in a Java source file. Any classes declared within that file will belong to thespecified package. The package statement defines a name space in which classes arestored. If you omit the package statement, the class names are put into the defaultpackage, which has no name.

This is the general form of the package statement:
package *pkg*;

Here, *pkg* is the name of the package. For example, the following statement creates apackage called MyPackage.

package MyPackage;

You can create a hierarchy of packages. To do so, simply separate each package namefrom the one above it by use of a period. The general form of a multileveled packagestatement is shown here:
package *pkg1*[.*pkg2*[.*pkg3*]];

Example:

```
String n, double b) {
name = n;
bal = b;
}
void show() {
if(bal<0)
System.out.print("—> ");
System.out.println(name + ": $" + bal);
}
}
```

```
class AccountBalance {
public static void main(String args[]) {
Balance current[] = new Balance[3];
current[0] = new Balance("K. J. Fielding", 123.23);
current[1] = new Balance("Will Tell", 157.02);
current[2] = new Balance("Tom Jackson", -12.33);
for(int i=0; i<3; i++) current[i].show();
}}
```

Access Protection :

Classes and packages are both means of encapsulating and containing the name spaceand scope of variables and methods. Packages act as containers for classes and othersubordinate packages. Classes act as containers for data and code. The class is Java'ssmallest unit of abstraction. Because of the interplay between classes and packages,Java addresses four categories of visibility for class members:

- Subclasses in the same package
- Non-subclasses in the same package
- Subclasses in different packages
- Classes that are neither in the same package nor subclasses

The three access specifiers, private, public, and protected, provide a variety of ways toproduce the many levels of access required by these categories.

Tabel 1.6

	Private	**No Modifier**	**Protected**	**Public**
Same Class	Yes	Yes	Yes	Yes
Same Package Subclass	No	Yes	Yes	Yes
Same Package Non-Subclass	No	Yes	Yes	Yes
Different Package Subclass	No	No	Yes	Yes
Different Package Non-Subclass	No	No	No	Yes

Interfaces :

Using interface, you can specify what a class must do, but nothow it does it. Interfaces are syntactically similar to classes, but they lack instancevariables, and their methods are declared without any body. To implement an interface, a class must create the complete set of methods defined bythe interface. However, each class is free to determine the details of its ownimplementation. By providing the interface keyword, Java allows you to fully utilize the"one interface, multiple methods" aspect of polymorphism.

Defining an Interface :

An interface is defined much like a class. This is the general form of an interface:

access interface name {
return-type method-name1(parameter-list);
return-type method-name2(parameter-list);
type final-varname1 = value;
type final-varname2 = value;
// ...
return-type method-nameN(parameter-list);
type final-varnameN = value;}

Here, *access* is either public or not used. When no access specifier is included, thendefault access results, and the interface is only available to other members of thepackage in which it is declared. When it is declared as public, the interface can be usedby any other code. *name* is the name of the interface, and can be any valid identifier.Notice that the methods which are declared have no bodies. They end with a semicolonafter the parameter list. They are, essentially, abstract methods; there can be no defaultimplementation of any method specified within an interface. Each class that includes aninterface must implement all of the methods.

Implementing Interfaces :

Once an interface has been defined, one or more classes can implement that interface.To implement an interface, include the implements clause in a class definition, and thencreate the methods defined by the interface. The general form of a class that includes theimplements clause looks like this:

access class *classname* [extends *superclass*]
[implements *interface* [,*interface...*]] {
// class-body
}

Here, *access* is either public or not used. If a class implements more than one interface,the interfaces are separated with a comma. If a class implements two interfaces thatdeclare the same method, then the same method will be used by clients of eitherinterface. The methods that implement an interface must be declared public. Also, thetype signature of the implementing method must match exactly the type signaturespecified in the interface definition.

1.9 MULTI-THREADING

A *thread* is defined as a path of execution, a collection of statements that execute in a specific order. Java provides built-in support for *multithreaded programming.* A multithreaded program contains two or more parts that can run concurrently. Each part of such a program is called a *thread,* and each thread defines a separate path of execution.

Life Cycle of a Thread:

A thread goes through various stages in its life cycle. For example, a thread isborn, started, runs, and then dies. Stages are:

Born.:

When a thread is first created, it is referred to as a *born* thread. Every thread has apriority, with a new thread inheriting the priority of the thread that created it. This priority can be changed at any time in the thread’s life cycle. Thread priority is an int value, and your Java threads can have any priority between 1 and 10. A born thread does not run until it is started.

Runnable :

After a newly born thread is started, the thread becomes *runnable*. For each of the 10 priorities, there is a corresponding priority queue (the first thread in is the first thread out). When a thread becomes runnable, it enters the queue of its respective priority

Running:

The thread scheduler determines when a runnable thread gets to actually run. In fact, the only way a thread is running is if the thread scheduler grants it permission. If for any reason a thread has to give up the CPU, it must eventually work its way through the runnable priority queues before it can run again.

Blocked :

A thread can become blocked, which occurs when multiple threads are synchronizing on the same data and need to take turns. A blocked thread is not running, nor is it runnable. It waits until the synchronization monitor allows it to continue, at which point it becomes runnable again and enters its appropriate priority queue.

Dead :

A thread that runs to completion is referred to as a dead thread. The term *dead* is used because it cannot be started again. If you need to repeat the task of the thread, you need to instantiate a new thread object.

Creating a Thread:

There are three common ways to write a thread in Java:

- You can write a class that implements the Runnable interface, then associatean instance of your class with a java.lang.Thread object.
- You can write a class that extends the Thread class.
- You can write a class that extends the java.util.TimerTask class, andthen schedule an instance of your class with a java.util.Timer object.

Either you write a class that implements Runnable, or you extend a class that already implements Runnable. (Both the Thread and TimerTask classes implement Runnable.) In either case, you define the one method in Runnable:

public void run()

The body of the run() method is the path of execution for your thread. Whenthe thread starts running, the run() method is invoked, and the threadbecomes dead when the run() method runs to completion.

Here is an example that creates a new thread and starts it running:

```
// Create a second thread.
class NewThread implements Runnable {
Thread t;
NewThread() {
// Create a new, second thread
t = new Thread(this, "Demo Thread");
System.out.println("Child thread: " + t);
t.start(); // Start the thread
}
// This is the entry point for the second thread.
public void run() {
try {
for(int i = 5; i > 0; i—) {
System.out.println("Child Thread: " + i);
Thread.sleep(500);
}
} catch (InterruptedException e) {
System.out.println("Child interrupted.");
}
System.out.println("Exiting child thread.");
}
}
class ThreadDemo {
public static void main(String args[]) {
new NewThread(); // create a new thread
try {
for(int i = 5; i > 0; i—) {
System.out.println("Main Thread: " + i);
Thread.sleep(1000);
}
} catch (InterruptedException e) {
System.out.println("Main thread interrupted.");
}
System.out.println("Main thread exiting.");
}
}
```

Creating Multiple Threads :

So far, you have been using only two threads: the main thread and one child thread.

However, your program can spawn as many threads as it needs. For example, thefollowing program creates three child threads:

```
// Create multiple threads.
class NewThread implements Runnable {
String name; // name of thread
Thread t;
NewThread(String threadname) {
name = threadname;
t = new Thread(this, name);
System.out.println("New thread: " + t);
t.start(); // Start the thread
}
// This is the entry point for thread.
public void run() {
try {
for(int i = 5; i > 0; i—) {
System.out.println(name + ": " + i);
Thread.sleep(1000);
}
} catch (InterruptedException e) {
System.out.println(name + "Interrupted");
}
System.out.println(name + " exiting.");
}
}
class MultiThreadDemo {
public static void main(String args[]) {
new NewThread("One"); // start threads
new NewThread("Two");
new NewThread("Three");
try {
// wait for other threads to end
Thread.sleep(10000);
} catch (InterruptedException e) {
System.out.println("Main thread Interrupted");
}
System.out.println("Main thread exiting.");
}
}
```

The output from this program is shown here:

```
New thread: Thread[One,5,main]
New thread: Thread[Two,5,main]
New thread: Thread[Three,5,main]
One: 5
Two: 5
Three: 5
One: 4
Two: 4
Three: 4
One: 3
Three: 3
Two: 3
One: 2
```

```
Three: 2
Two: 2
One: 1
Three: 1
Two: 1
One exiting.
Two exiting.
Three exiting.
Main thread exiting.
```

1.10 MANAGING ERRORS AND EXCEPTIONS

This chapter examines Java's exception-handling mechanism. An *exception* is anabnormal condition that arises in a code sequence at run time. In other words,an exception is a run-time error. In computer languages that do not supportexception handling, errors must be checked and handled manually—typically throughthe use of error codes, and so on. This approach is as cumbersome as it is troublesome.Java's exception handling avoids these problems and, in the process, brings run-timeerror management into the object-oriented world.For the most part, exception handling has not changed since the original versionof Java. However, Java 2, version 1.4 has added a new subsystem called the *chainedexception facility*. This feature is described near the end of this chapter.

Exception-Handling Fundamentals :

A Java exception is an object that describes an exceptional (that is, error) conditionthat has occurred in a piece of code. When an exceptional condition arises, an objectrepresenting that exception is created and *thrown* in the method that caused the error.That method may choose to handle the exception itself, or pass it on. Either way, atsome point, the exception is *caught* and processed. Exceptions can be generated by theJava run-time system, or they can be manually generated by your code. Exceptionsthrown by Java relate to fundamental errors that violate the rules of the Java languageor the constraints of the Java execution environment. Manually generated exceptionsare typically used to report some error condition to the caller of a method.

Java exception handling is managed via five keywords: try, catch, throw, throws,and finally. Briefly, here is how they work. Program statements that you want tomonitor for exceptions are contained within a try block. If an exception occurs withinthe try block, it is thrown. Your code can catch this exception (using catch) and handleit in some rational manner. System-generated exceptions are automatically thrown bythe Java run-time system. To manually throw an exception, use the keyword throw.Any exception that is thrown out of a method must be specified as such by a throwsclause. Any code that absolutely must be executed before a method returns is put ina finally block.

This is the general form of an exception-handling block:

```
try {
// block of code to monitor for errors}
catch (ExceptionType1 exOb) {
// exception handler for ExceptionType1
}
catch (ExceptionType2 exOb) {
// exception handler for ExceptionType2
}
// ...
finally {
// block of code to be executed before try block ends
}
```

Here, *ExceptionType* is the type of exception that has occurred. The remainder of this

chapter describes how to apply this framework.

Exception Types :

All exception types are subclasses of the built-in class Throwable. Thus, Throwableis at the top of the exception class hierarchy. Immediately below Throwable are twosubclasses that partition exceptions into two distinct branches. One branch is headedby Exception. This class is used for exceptional conditions that user programs shouldcatch. This is also the class that you will subclass to create your own custom exceptiontypes. There is an important subclass of Exception, called RuntimeException.Exceptions of this type are automatically defined for the programs that you writeand include things such as division by zero and invalid array indexing.The other branch is topped by Error, which defines exceptions that are not expectedto be caught under normal circumstances by your program. Exceptions of type Errorare used by the Java run-time system to indicate errors having to do with the run-timeenvironment, itself. Stack overflow is an example of such an error. This chapter willnot be dealing with exceptions of type Error, because these are typically created inresponse to catastrophic failures that cannot usually be handled by your program.

Uncaught Exceptions :

Before you learn how to handle exceptions in your program, it is useful to see whathappens when you don’t handle them. This small program includes an expression thatintentionally causes a divide-by-zero error.

```
class Exc0 {
public static void main(String args[]) {
int d = 0;
int a = 42 / d;
}
}
```

When the Java run-time system detects the attempt to divide by zero, it constructs anew exception object and then *throws* this exception. This causes the execution of Exc0to stop, because once an exception has been thrown, it must be *caught* by an exceptionhandler and dealt with immediately. In this example, we haven't supplied any exceptionhandlers of our own, so the exception is caught by the default handler provided by theJava run-time system. Any exception that is not caught by your program will ultimatelybe processed by the default handler. The default handler displays a string describingthe exception, prints a stack trace from the point at which the exception occurred, andterminates the program.

Here is the output generated when this example is executed.

```
java.lang.ArithmeticException: / by zeroat
Exc0.main(Exc0.java:4)
```

Notice how the class name, Exc0; the method name, main; the filename, Exc0.java;and the line number, 4, are all included in the simple stack trace. Also, notice that thetype of the exception thrown is a subclass of Exception called ArithmeticException,which more specifically describes what type of error happened. As discussed later inthis chapter, Java supplies several built-in exception types that match the various sortsof run-time errors that can be generated.The stack trace will always show the sequence of method invocations that led up tothe error. For example, here is another version of the preceding program that introducesthe same error but in a method separate from main( ):

```
class Exc1 {
static void subroutine() {
int d = 0;
int a = 10 / d;
}
public static void main(String args[]) {
Exc1.subroutine();
}
}
```

The resulting stack trace from the default exception handler shows how the entirecall stack is displayed:

```
java.lang.ArithmeticException: / by zero
at Exc1.subroutine(Exc1.java:4)
at Exc1.main(Exc1.java:7)
```

As you can see, the bottom of the stack is main’s line 7, which is the call tosubroutine( ), which caused the exception at line 4. The call stack is quite useful fordebugging, because it pinpoints the precise sequence of steps that led to the error.

Using try and catch :

Although the default exception handler provided by the Java run-time system is usefulfor debugging, you will usually want to handle an exception yourself. Doing soprovides two benefits. First, it allows you to fix the error. Second, it prevents theprogram from automatically terminating. Most users would be confused (to say theleast) if your program stopped running and printed a stack trace whenever an erroroccurred! Fortunately, it is quite easy to prevent this.

To guard against and handle a run-time error, simply enclose the code that youwant to monitor inside a try block. Immediately following the try block, include a catchclause that specifies the exception type that you wish to catch. To illustrate how easilythis can be done, the following program includes a try block and a catch clause whichprocesses the ArithmeticException generated by the division-by-zero error:

```
class Exc2 {
public static void main(String args[]) {
int d, a;
try { // monitor a block of code.
d = 0;
a = 42 / d;
System.out.println("This will not be printed.");
} catch (ArithmeticException e) { // catch divide-by-zero error
System.out.println("Division by zero.");
}
System.out.println("After catch statement.");
}
}
```

This program generates the following output:

```
Division by zero.
After catch statement.
```

Notice that the call to println( ) inside the try block is never executed. Once anexception is thrown, program control transfers out of the try block into the catch block.Put differently, catch is not “called,” so execution never “returns” to the try block froma catch. Thus, the line “This will not be printed.” is not displayed. Once the catchstatement has executed, program control continues with the next line in the programfollowing the entire try/catch mechanism.A try and its catch statement form a unit. The scope of the catch clause is restrictedto those statements specified by the immediately preceding try statement. A catchstatement cannot catch an exception thrown by another try statement (except in thecase of nested try statements, described shortly). The statements that are protected bytry must be surrounded by curly braces. (That is, they must be within a block.) Youcannot use try on a single statement.The goal of most well-constructed catch clauses should be to resolve theexceptional condition and then continue on as if the error had never happened.

For example, in the next program each iteration of the for loop obtains two randomintegers. Those two integers are divided by each other, and the result is used to dividethe value 12345. The final result is put into a. If either division operation causes adivide-by-zero error, it is caught, the value of a is set to zero, and the programcontinues.

```
// Handle an exception and move on.
import java.util.Random;
class HandleError {
public static void main(String args[]) {
int a=0, b=0, c=0;
Random r = new Random();
for(int i=0; i<32000; i++) {
try {
b = r.nextInt();
c = r.nextInt();
a = 12345 / (b/c);
} catch (ArithmeticException e) {
System.out.println("Division by zero.");
a = 0; // set a to zero and continue
}
System.out.println("a: " + a);
}
}
}
```

Displaying a Description of an Exception :

Throwable overrides the toString( ) method(defined by Object) so that it returns astring containing a description of the exception. You can display this description in aprintln( ) statement by simply passing the exception as an argument. For example, thecatch block in the preceding program can be rewritten like this:

```
catch (ArithmeticException e) {
System.out.println("Exception: " + e);
a = 0; // set a to zero and continue
}
```

When this version is substituted in the program, and the program is run, eachdivide-by-zero error displays the following message:

```
Exception: java.lang.ArithmeticException: / by zero
```

While it is of no particular value in this context, the ability to display a descriptionof an exception is valuable in other circumstances—particularly when you areexperimenting with exceptions or when you are debugging.

Multiple catch Clauses:

In some cases, more than one exception could be raised by a single piece of code. Tohandle this type of situation, you can specify two or more catch clauses, each catchinga different type of exception. When an exception is thrown, each catch statement isinspected in order, and the first one whose type matches that of the exception isexecuted. After one catch statement executes, the others are bypassed, and executioncontinues after the try/catch block. The following example traps two differentexception types:

```
// Demonstrate multiple catch statements.
class MultiCatch {
public static void main(String args[]) {
try {
int a = args.length;
System.out.println("a = " + a);
int b = 42 / a;
int c[] = { 1 };
c[42] = 99;
} catch(ArithmeticException e) {
System.out.println("Divide by 0: " + e);
} catch(ArrayIndexOutOfBoundsException e) {
System.out.println("Array index oob: " + e);
}
System.out.println("After try/catch blocks.");
}
}
```

This program will cause a division-by-zero exception if it is started with no commandlineparameters, since a will equal zero. It will survive the division if you provide acommand-line argument, setting a to something larger than zero. But it will cause anArrayIndexOutOfBoundsException, since the int array c has a length of 1, yet theprogram attempts to assign a value to c[42].

Here is the output generated by running it both ways:

```
C:\>java MultiCatch
a = 0
Divide by 0: java.lang.ArithmeticException: / by zero
After try/catch blocks.
C:\>java MultiCatch TestArg
a = 1

Array index oob: java.lang.ArrayIndexOutOfBoundsException
After try/catch blocks.
```

When you use multiple catch statements, it is important to remember thatexception subclasses must come before any of their superclasses. This is because acatch statement that uses a superclass will catch exceptions of that type plus any ofits subclasses. Thus, a subclass would never be reached if it came after its superclass.Further, in Java, unreachable code is an error. For example, consider the followingprogram:

```
/* This program contains an error.
A subclass must come before its superclass in
a series of catch statements. If not,
unreachable code will be created and a
compile-time error will result.
*/
class SuperSubCatch {
public static void main(String args[]) {
try {
int a = 0;
int b = 42 / a;
} catch(Exception e) {
System.out.println("Generic Exception catch.");
}
/* This catch is never reached because
ArithmeticException is a subclass of Exception. */
catch(ArithmeticException e) { // ERROR - unreachable
System.out.println("This is never reached.");
}
}
}
```

If you try to compile this program, you will receive an error message stating thatthe second catch statement is unreachable because the exception has already beencaught. Since ArithmeticException is a subclass of Exception, the first catch statementwill handle all Exception-based errors, including ArithmeticException. This meansthat the second catch statement will never execute. To fix the problem, reverse theorder of the catch statements.

Nested try Statements :

The try statement can be nested. That is, a try statement can be inside the block ofanother try. Each time a try statement is entered, the context of that exception ispushed on the stack. If an inner try statement does not have a catch handler for aparticular exception, the stack is unwound and the next try statement’s catch handlersare inspected for a match. This continues until one of the catch statements succeeds, oruntil all of the nested try statements are exhausted. If no catch statement matches, thenthe Java run-time system will handle the exception. Here is an example that usesnested try statements:

```
// An example of nested try statements.
class NestTry {
public static void main(String args[]) {
try {
int a = args.length;
/* If no command-line args are present,
the following statement will generate
a divide-by-zero exception. */
int b = 42 / a;
System.out.println("a = " + a);
try { // nested try block
/* If one command-line arg is used,
then a divide-by-zero exception
will be generated by the following code. */
if(a==1) a = a/(a-a); // division by zero
/* If two command-line args are used,
then generate an out-of-bounds exception. */
if(a==2) {
int c[] = { 1 };
c[42] = 99; // generate an out-of-bounds exception
}
} catch(ArrayIndexOutOfBoundsException e) {
System.out.println("Array index out-of-bounds: " + e);
}
} catch(ArithmeticException e) {
System.out.println("Divide by 0: " + e);
}
}
}
```

As you can see, this program nests one try block within another. The programworks as follows. When you execute the program with no command-line arguments, adivide-by-zero exception is generated by the outer try block. Execution of the programby one command-line argument generates a divide-by-zero exception from within thenested try block. Since the inner block does not catch this exception, it is passed onto the outer try block, where it is handled. If you execute the program with twocommand-line arguments, an array boundary exception is generated from withinthe inner try block. Here are sample runs that illustrate each case:

```
C:\>java NestTry
Divide by 0: java.lang.ArithmeticException: / by zero
C:\>java NestTry One
a = 1
Divide by 0: java.lang.ArithmeticException: / by zero
C:\>java NestTry One Two
a = 2
```

Array index out-of-bounds:

java.lang.ArrayIndexOutOfBoundsException

Nesting of try statements can occur in less obvious ways when method calls areinvolved. For example, you can enclose a call to a method within a try block. Insidethat method is another try statement. In this case, the try within the method is stillnested inside the outer try block, which calls the method. Here is the previous programrecoded so that the nested try block is moved inside the method nesttry( ):

```
/* Try statements can be implicitly nested via
calls to methods. */
class MethNestTry {
static void nesttry(int a) {
try { // nested try block
/* If one command-line arg is used,
then a divide-by-zero exception
will be generated by the following code. */
if(a==1) a = a/(a-a); // division by zero
/* If two command-line args are used,
then generate an out-of-bounds exception. */
if(a==2) {
int c[] = { 1 };
c[42] = 99; // generate an out-of-bounds exception
}
} catch(ArrayIndexOutOfBoundsException e) {
System.out.println("Array index out-of-bounds: " + e);
}
}
public static void main(String args[]) {
try {
int a = args.length;
/* If no command-line args are present,
the following statement will generate
a divide-by-zero exception. */
int b = 42 / a;
System.out.println("a = " + a);
nesttry(a);
} catch(ArithmeticException e) {
System.out.println("Divide by 0: " + e);
}
}
}
```

The output of this program is identical to that of the preceding example.

Throw:

So far, you have only been catching exceptions that are thrown by the Java run-timesystem. However, it is possible for your program to throw an exception explicitly,using the throw statement. The general form of throw is shown here:throw *ThrowableInstance*;

Here, *ThrowableInstance* must be an object of type Throwable or a subclass ofThrowable. Simple types, such as int or char, as well as non-Throwable classes, suchas String and Object, cannot be used as exceptions. There are two ways you can obtaina Throwable object: using a parameter into a catch clause, or creating one with the newoperator.The flow of execution stops immediately after the throw statement; any subsequentstatements are not executed. The nearest enclosing try block is inspected to see if it hasa catchstatement that matches the type of the exception. If it does find a match, controlis transferred to that statement. If not, then the next enclosing try statement isinspected, and so on. If no matching catch is found, then the default exception handlerhalts the program and prints the stack trace.

Here is a sample program that creates and throws an exception. The handler thatcatches the exception rethrows it to the outer handler.

```
// Demonstrate throw.
class ThrowDemo {
static void demoproc() {
try {
throw new NullPointerException("demo");
} catch(NullPointerException e) {
System.out.println("Caught inside demoproc.");
throw e; // rethrow the exception
}
}
public static void main(String args[]) {
try {
demoproc();
} catch(NullPointerException e) {
System.out.println("Recaught: " + e);
}
}
}
```

This program gets two chances to deal with the same error. First, main( ) sets up anexception context and then calls demoproc( ). The demoproc( ) method then sets upanother exception-handling context and immediately throws a new instance ofNullPointerException, which is caught on the next line. The exception is thenrethrown. Here is the resulting output:Caught inside demoproc.

Recaught: java.lang.NullPointerException: demo

The program also illustrates how to create one of Java's standard exception objects.

Pay close attention to this line:

```
throw new NullPointerException("demo");
```

Here, new is used to construct an instance of NullPointerException. All of Java'sbuilt-in run-time exceptions have at least two constructors: one with no parameterand one that takes a string parameter. When the second form is used, the argumentspecifies a string that describes the exception. This string is displayed when the objectis used as an argument to print( ) or println( ). It can also be obtained by a call togetMessage( ), which is defined by Throwable.

Throws :

If a method is capable of causing an exception that it does not handle, it must specifythis behavior so that callers of the method can guard themselves against that exception.You do this by including a throws clause in the method's declaration. A throws clauselists the types of exceptions that a method might throw. This is necessary for allexceptions, except those of type Error or RuntimeException, or any of their subclasses.All other exceptions that a method can throw must be declared in the throws clause. Ifthey are not, a compile-time error will result.
This is the general form of a method declaration that includes a throws clause:

```
type method-name(parameter-list) throws exception-list
{
// body of method
}
```

Here, *exception-list* is a comma-separated list of the exceptions that a method can throw.

Following is an example of an incorrect program that tries to throw an exceptionthat it does not catch. Because the program does not specify a throws clause to declarethis fact, the program will not compile.

```
// This program contains an error and will not compile.
class ThrowsDemo {
static void throwOne() {
System.out.println("Inside throwOne.");
throw new IllegalAccessException("demo");
}
public static void main(String args[]) {
throwOne();
}
}
```

To make this example compile, you need to make two changes. First, you need todeclare that throwOne( ) throws IllegalAccessException. Second, main( ) must definea try/catch statement that catches this exception.The corrected example is shown here:

```
// This is now correct.
class ThrowsDemo {
static void throwOne() throws IllegalAccessException {
System.out.println("Inside throwOne.");
throw new IllegalAccessException("demo");
}
public static void main(String args[]) {
try {
throwOne();
} catch (IllegalAccessException e) {
System.out.println("Caught " + e);
}
}
}
```

Here is the output generated by running this example program:

```
inside throwOne

caught java.lang.IllegalAccessException: demo
```

finally :

When exceptions are thrown, execution in a method takes a rather abrupt, nonlinearpath that alters the normal flow through the method. Depending upon how themethod is coded, it is even possible for an exception to cause the method to returnprematurely. This could be a problem in some methods. For example, if a methodopens a file upon entry and closes it upon exit, then you will not want the code thatcloses the file to be bypassed by the exception-handling mechanism. The finallykeyword is designed to address this contingency.finally creates a block of code that will be executed after a try/catch block hascompleted and before the code following the try/catch block. The finally block willexecute whether or not an exception is thrown. If an exception is thrown, the finallyblock will execute even if no catch statement matches the exception. Any time amethod is about to return to the caller from inside a try/catch block, via an uncaughtexception or an explicit return statement, the finally clause is also executed just beforethe method returns. This can be useful for closing file handles and freeing up any otherresources that might have been allocated at the beginning of a method with the intentof disposing of them before returning. The finally clause is optional. However, each trystatement requires at least one catch or a finally clause.

Here is an example program that shows three methods that exit in various ways,

none without executing their finally clauses:

```
// Demonstrate finally.
class FinallyDemo {
// Through an exception out of the method.
static void procA() {
try {
System.out.println("inside procA");
throw new RuntimeException("demo");
} finally {
System.out.println("procA's finally");
}
}
// Return from within a try block.
static void procB() {
try {
System.out.println("inside procB");
return;
} finally {
System.out.println("procB's finally");
}
}
// Execute a try block normally.
static void procC() {
try {
System.out.println("inside procC");
} finally {
System.out.println("procC's finally");
}
}
public static void main(String args[]) {
try {
procA();
} catch (Exception e) {
System.out.println("Exception caught");
}
procB();
procC();
}
}
```

In this example, procA( ) prematurely breaks out of the try by throwing anexception. The finally clause is executed on the way out. procB( )’s try statement isexited via a return statement. The finally clause is executed before procB( ) returns. InprocC( ), the try statement executes normally, without error. However, the finallyblock is still executed.

If a finally block is associated with a try, the finally block will be executed upon conclusion of the try.

Here is the output generated by the preceding program:

```
inside procA
procA’s finally
Exception caught
inside procB
procB’s finally
inside procC
procC’s finally
```

Java’s Built-in Exceptions :

Inside the standard package java.lang, Java defines several exception classes. A fewhave been used by the preceding examples. The most general of these exceptionsare subclasses of the standard type RuntimeException. Since java.lang is implicitlyimported into all Java programs, most exceptions derived from RuntimeExceptionare automatically available. Furthermore, they need not be included in any method’sthrows list. In the language of Java, these are called *unchecked exceptions* because thecompiler does not check to see if a method handles or throws these exceptions. Theunchecked exceptions defined in java.lang are listed in Table-1. Table-2 lists thoseexceptions defined by java.lang that must be included in a method’s throws list if thatmethod can generate one of these exceptions and does not handle it itself. These arecalled *checked exceptions.* Java defines several other types of exceptions that relate to itsvarious class libraries.

Tabel 1.7

Exception	**Meaning**
ArithmeticException	Arithmetic error, such as divide-by-zero.
ArrayIndexOutOfBoundsException	Array index is out-of-bounds.
ArrayStoreException	Assignment to an array element of an incompatible type.
ClassCastException	Invalid cast.
IllegalArgumentException	Illegal argument used to invoke amethod.
IllegalMonitorStateException	Illegal monitor operation, such as waiting on an unlocked thread.
IllegalStateException	Environment or application is inincorrect state.

IllegalThreadStateException	Requested operation not compatiblewith current thread state.
IndexOutOfBoundsException	Some type of index is out-of-bounds.
NegativeArraySizeException	Array created with a negative size.
ClassNotFoundException	Class not found.
CloneNotSupportedException	Attempt to clone an object that does notimplement the Cloneable interface.
IllegalAccessException	Access to a class is denied. InstantiationException Attempt to create an object of anabstract class or interface.
InterruptedException	One thread has bee interrupted by another thread.
NoSuchFieldException	A requested field does not exist.
NoSuchMethodException	A requested method does not exist.
NullPointerException	Invalid use of a null reference.
NumberFormatException	Invalid conversion of a string to anumeric format.
SecurityException	Attempt to violate security.
StringIndexOutOfBounds	Attempt to index outside the bounds ofa string.
UnsupportedOperationException	An unsupported operation Wasencountered.

Creating Your Own Exception Subclasses:

Although Java’s built-in exceptions handle most common errors, you will probably wantto create your own exception types to handle situations specific to your applications.This is quite easy to do: just define a subclass of Exception (which is, of course, a subclassof Throwable). Your subclasses don’t need to actually implement anything—it is theirexistence in the type system that allows you to use them as exceptions.The Exception class does not define any methods of its own. It does, of course,inherit those methods provided by Throwable. Thus, all exceptions, including thosethat you create, have the methods defined by Throwable available to them. They areshown in Table-3. Notice that several methods were added by Java 2, version 1.4.

You may also wish to override one or more of these methods in exception classes thatyou create.

Tabel 1.8

Method	Description
Throwable fillInStackTrace( )	Returns a Throwable object that containsa completed stack trace. This object can berethrown.
Throwable getCause( )	Returns the exception that underlies thecurrent exception.If there is no underlyingexception, null is returned. Added by Java 2,version 1.4.
String getLocalizedMessage( )	Returns a localized description of theexception.
String getMessage( )	Returns a description of the exception.
StackTraceElement[ ] getStackTrace( )	Returns an array that contains the stacktrace, oneelement at a time as an array ofStackTraceElement. The method at the topof the stack is the last method called beforethe exception was thrown. This methodis found in the first element of the array.The StackTraceElement class gives yourprogram access to information about eachelement in the trace, such as its methodname. Added by Java 2, version 1.4
Throwable initCause (Throwable*causeExc*)	Associates *causeExc* with the invokingexception as a cause of the invoking exception.Returns a reference to the exception. Addedby Java 2, version 1.4
void printStackTrace( )	Displays the stack trace.
void printStackTrace(PrintStream *stream*)	Sends the stack trace to the specified stream.
void printStackTrace(PrintWriter *stream*)	Sends the stack trace to the specified stream.
void setStackTrace (StackTraceElement *elements*[ ])	Sets the stack trace to the elements passedin *elements*.

	This method is for specializedapplications, not normal use. Added by Java 2,version 1.4
String toString( )	Returns a String object containing adescription of the exception. This methodis called byprintln( ) whenoutputting aThrowable object.

The following example declares a new subclass of Exception and then uses thatsubclass to signal an error condition in a method. It overrides the toString( ) method,allowing the description of the exception to be displayed using println( ).

```
// This program creates a custom exception type.
class MyException extends Exception {
private int detail;
MyException(int a) {
detail = a;
}
public String toString() {
return "MyException[" + detail + "]";
}
}
class ExceptionDemo {
static void compute(int a) throws MyException {
System.out.println("Called compute(" + a + ")");
if(a > 10)
throw new MyException(a);
System.out.println("Normal exit");
}
public static void main(String args[]) {
try {
compute(1);
compute(20);
} catch (MyException e) {
System.out.println("Caught " + e);
}
}
}
```

This example defines a subclass of Exception called MyException. This subclassis quite simple: it has only a constructor plus an overloaded toString( ) method thatdisplays the value of the exception. The ExceptionDemo class defines a methodnamed compute( ) that throws a MyException object. The exception is thrown whencompute( )’s integer parameter is greater than 10. The main( ) method sets up anexception handler for MyException, then calls compute() with a legal value (lessthan 10) and an illegal one to show both paths through the code. Here is the result:

```
Called compute(1)
Normal exit
Called compute(20)
Caught MyException[20]
```

Chained Exceptions :

Java 2, version 1.4 added a new feature to the exception subsystem: *chained exceptions*.The chained exception feature allows you to associate another exception with an exception.This second exception describes the cause of the first exception. For example, imagine asituation in which a method throws an ArithmeticException because of an attempt todivide by zero. However, the actual cause of the problem was that an I/O error occurred,which caused the divisor to be set improperly. Although the method must certainly throwan ArithmeticException, since that is the error that occurred, you might also want to letthe calling code know that the underlying cause was an I/O error. Chained exceptionslet you handle this, and any other situation in which layers of exceptions exist.To allow chained exceptions, Java 2, version 1.4 added two constructors and twomethods to Throwable. The constructors are shown here.

Throwable(Throwable*causeExc*)Throwable(String *msg*, Throwable *causeExc*)

In the first form, *causeExc* is the exception that causes the current exception. That is,*causeExc* is the underlying reason that an exception occurred. The second form allowsyou to specify a description at the same time that you specify a cause exception. Thesetwo constructors have also been added to the Error, Exception, and RuntimeExceptionclasses.The chained exception methods added to Throwable are getCause( ) and initCause( ).These methods are shown in Table-3, and are repeated here for the sake of discussion.

Throwable getCause( )
Throwable initCause(Throwable *causeExc*)

The getCause( ) method returns the exception that underlies the current exception.If there is no underlying exception, null is returned. The initCause( ) method associates*causeExc* with the invoking exception and returns a reference to the exception. Thus, youcan associate a cause with an exception after the exception has been created. However, thecause exception can be set only once. Thus, you can call initCause( ) only once for eachexception object. Furthermore, if the cause exception was set by a constructor, then youcan’t set it again using initCause( ).In general, initCause( ) is used to set a cause for legacy exception classes whichdon’t support the two additional constructors described earlier. At the time of thiswriting, most of Java’s built-in exceptions, such as ArithmeticException, do not definethe additional constructors. Thus, you will use initCause( ) if you need to add anexception chain to these exceptions. When creating your own exception classes youwill want to add the two chained-exception constructors if you will be using yourexceptions in situations in which layered exceptions are possible.Here is an example that illustrates the mechanics of handling chained exceptions.

```
// Demonstrate exception chaining.
class ChainExcDemo {
static void demoproc() {
// create an exception
NullPointerException e =
new NullPointerException("top layer");
// add a cause
e.initCause(new ArithmeticException("cause"));
throw e;
}
public static void main(String args[]) {
try {
demoproc();
} catch(NullPointerException e) {
// display top level exception
System.out.println("Caught: " + e);
// display cause exception
System.out.println("Original cause: " +
e.getCause());
}
}
}
```

The output from the program is shown here.

```
Caught: java.lang.NullPointerException: top layer
Original cause: java.lang.ArithmeticException: cause
```

In this example, the top-level exception is NullPointerException. To it is addeda cause exception, ArithmeticException. When the exception is thrown out ofdemoproc( ), it is caught by main( ). There, the top-level exception is displayed,followed by the underlying exception, which is obtained by calling getCause( ).Chained exceptions can be carried on to whatever depth is necessay. Thus, thecause exception can, itself, have a cause. Be aware that overly long chains of exceptionsmay indicate poor design.Chained exceptions are not something that every program will need. However, incases in which knowledge of an underlying cause is useful, they offer an elegant solution.

Using Exceptions

Exception handling provides a powerful mechanism for controlling complex programsthat have many dynamic run-time characteristics. It is important to think of try, throw,and catch as clean ways to handle errors and unusual boundary conditions in yourprogram's logic. If you are like most programmers, then you probably are used toreturning an error code when a method fails. When you are programming in Java, youshould break this habit. When a method can fail, have it throw an exception. This is acleaner way to handle failure modes.One last point: Java's exception-handling statements should not be considered ageneral mechanism for nonlocal branching. If you do so, it will only confuse your codeand make it hard to maintain.

1.11 THE APPLET PROGRAMMING

This chapter examines the Applet class, which provides the necessary support forapplets. In Chapter 12, you were introduced to the general form of an applet andthe steps necessary to ompile and run one. In this chapter, we will look atapplets in detail.The Applet class is contained in the java.applet package. Applet contains severalmethods that give you detailed control over the execution of your applet. In addition,java.applet also defines three interfaces: AppletContext, AudioClip, and AppletStub.Let's begin by reviewing the basic elements of an applet and the steps necessary tocreate and test one.

Applet Basics :

All applets are subclasses of Applet. Thus, all applets must import java.applet. Appletsmust also import java.awt. Recall that AWT stands for the Abstract Window Toolkit.Since all applets run in a window, it is necessary to include support for that window.Applets are not executed by the console-based Java run-time interpreter. Rather, theyare executed by either a Web browser or an applet viewer. The figures shown in thischapter were created with the standard applet viewer, called appletviewer, providedby the SDK. But you can use any applet viewer or browser you like.Execution of an applet does not begin at main( ). Actually, few applets even havemain( ) methods. Instead, execution of an applet is started and controlled with anentirely different mechanism, which will be explained shortly.

Output to your applet’swindow is not performed by System.out.println( ). Rather, it is handled with variousAWT methods, such as drawString( ), which outputs a string to a specified X,Ylocation. Input is also handled differently than in an application.Once an applet has been compiled, it is included in an HTML file using theAPPLET tag. The applet will be executed by a Java-enabled web browser when itencounters the APPLET tag within the HTML file. To view and test an applet moreconveniently, simply include a comment at the head of your Java source code file thatcontains the APPLET tag. This way, your code is documented with the necessaryHTML statements needed by your applet, and you can test the compiled applet bystarting the applet viewer with your Java source code file specified as the target. Hereis an example of such a comment:

```
/*
<applet code="MyApplet" width=200 height=60>
</applet>
*/
```

This comment contains an APPLET tag that will run an applet called MyApplet in awindow that is 200 pixels wide and 60 pixels high. Since the inclusion of an APPLETcommand makes testing applets easier, all of the applets shown in this book will contain the appropriate APPLET tag embedded in a comment.

The Applet Class :

The Applet class defines the methods shown in Table 19-1. Applet provides allnecessary support for applet execution, such as starting and stopping. It also providesmethods that load and display images, and methods that load and play audio clips.Applet extends the AWT class Panel. In turn, Panel extends Container, which extendsComponent. These classes provide support for Java’s window-based, graphicalinterface. Thus, Applet provides all of the necessary support for window-basedactivities. (The AWT is described in detail in following chapters.)

Tabel 1.9

Method	Description
void destroy( )	Called by the browser just beforeanapplet isterminated.Yourapplet will override this method if itneeds to perform any cleanup prior toIts destruction.
AccessibleContext	

getAccessibleContext( )	Returns the accessibilty context for theinvoking object.
AppletContext	getAppletContext( ) Returns the context associated with
the applet.	
String getAppletInfo( )	Returns a string that describes
the applet.	
AudioClip getAudioClip(URL *url*)	Returns an AudioClip object thatencapsulates the audio clip found at thelocation specified by *url.*
AudioClip getAudioClip(URL *url*, String c*lipName*)	Returns an AudioClip object thatencapsulates the audio clip found at thelocation specified by *url* and having thename specified by *clipName.*
URL getCodeBase( )	Returns the URL associated with theinvoking applet.
URL getDocumentBase( )	Returns the URL of the HTML
document that invokes the applet.	
Image getImage(URL *url*)	Returns an Image object thatencapsulates the image found at thelocation specified by *url.*
Image getImage(URL *url*, String *imageName*)	Returns an Image object that encapsulates the image found at thelocation specified by *url* and having thename specified by *imageName.*
Locale getLocale( )	Returns a Locale object that is usedby various locale-sensitive classes
and methods.	
String getParameter (String *paramName*)	Returns the parameter associated with*paramName.* null is returned if thespecified parameter is not found.
String[ ] [ ] getParameterInfo( )	Returns a String table that describesthe parameters recognized by theapplet. Each entry in the table mustconsist of three strings

	that contain thename of theparameter, a description ofits type and/or range, and anexplanation of its purpose.
void init( )	Called when an applet beginsexecution. It is the first method called
for any applet.	
boolean isActive( )	Returns true if the applet has beenstarted. It returns false if the applet hasbeen stopped.
static final AudioClip newAudioClip(URL *url*)	Returns an AudioClip object thatencapsulates the audio clip found at thelocation specified by *url*. This method issimilar to getAudioClip( ) except that itis static and can be executed withoutthe need for an Applet object. (Added nby Java 2)
void play(URL *url*)	If an audio clip is found at the locationspecified by *url,* the clip is played.
void play(URL *url*, String *clipName*)	If an audio clip is found at the locationspecified by *url* with the name specifiedby *clipName,* the clip is played.
void resize(Dimension *dim*)	Resizes the applet according to thedimensions specified by *dim*.Dimension is a class stored insidejava.awt. It contains two integer fields:width and height.
void resize(int *width*, int *height*)	Resizes the applet according tothe dimensions specified by *width*and *height.*
final void setStub (AppletStub *stubObj*)	Makes *stubObj* the stub for the applet.This method is used by the run-timesystem and is not usually called byyour applet. A *stub* is a small piece ofcode that

	provides the linkage betweenyour applet and the browser.
void showStatus(String *str*)	Displays *str* in the status window of thebrowser or applet viewer. If thebrowser does not support a statuswindow, then no action takes place.
void start( )	Called by the browser when an appletshould start (or resume) execution. It isautomatically called after init( ) whenan applet first begins.
void stop( )	Called by the browser to suspend execution of the applet.
calls start( ).	Once stopped,an applet is restarted when the browser

Applet Architecture :

An applet is a window-based program. As such, its architecture is different from theso-called normal, console-based programs shown in the first part of this book. If youare familiar with Windows programming, you will be right at home writing applets.If not, then there are a few key concepts you must understand.First, applets are event driven. Although we won’t examine event handling untilthe following chapter, it is important to understand in a general way how theevent-driven architecture impacts the design of an applet. An applet resembles a setof interrupt service routines. Here is how the process works. An applet waits until anevent occurs. The AWT notifies the applet about an event by calling an event handlerthat has been provided by the applet. Once this happens, the applet must takeappropriate action and then quickly return control to the AWT. This is a crucial point.For the most part, your applet should not enter a “mode” of operation in which itmaintains control for an extended period. Instead, it must perform specific actionsin response to events and then return control to the AWT run-time system. In thosesituations in which your applet needs to perform a repetitive task on its own (forexample, displaying a scrolling message across its window), you must start anadditional thread of execution. (You will see an example later in this chapter.)Second, the user initiates interaction with an applet—not the other way around. Asyou know, in a nonwindowed program, when the program needs input, it will promptthe user and then call some input method, such as readLine( ). This is not the way itworks in an applet. Instead, the user interacts with the applet as he or she wants, whenhe or she wants. These interactions are sent to the applet as events to which the appletmust respond. For example, when the user clicks a mouse inside the applet's window,a mouse-clicked event is generated. If the user presses a key while the applet's windowhas input focus, a keypress event is generated. As you will see in later chapters, appletscan contain arious controls, such as push buttons and check boxes. When the userinteracts with one of these ontrols, an event is generated.While the architecture of an applet is not as easy to understand as that of aconsole-based program, Java's AWT makes it as simple as possible. If you havewritten programs for Windows, you know how intimidating that environment canbe. Fortunately, Java's AWT provides a much cleaner approach that is morequickly mastered.

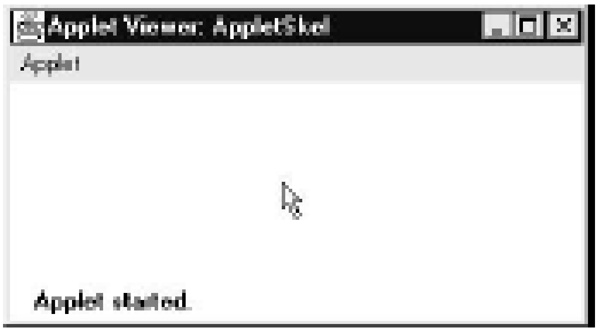
An Applet Skeleton :

All but the most trivial applets override a set of methods that provides the basicmechanism by which the browser or applet viewer interfaces to the applet and controlsits execution. Four of these methods—init( ), start( ), stop( ), and destroy( )—aredefined by Applet. Another, paint( ), is defined by the AWT Component class. Defaultimplementations for all of these methods are provided. Applets do not need tooverride those methods they do not use. However, only very simple applets will notneed to define all of them. These five methods can be assembled into the skeletonshown here:

```
// An Applet skeleton.
import java.awt.*;
import java.applet.*;
/*
<applet code="AppletSkel" width=300 height=100>
</applet>
*/
public class AppletSkel extends Applet {
// Called first.
public void init() {
// initialization
}
/* Called second, after init(). Also called whenever
the applet is restarted. */
public void start() {
// start or resume execution
}
// Called when the applet is stopped.
public void stop() {
// suspends execution
}
/* Called when applet is terminated. This is the last
method executed. */
public void destroy() {
// perform shutdown activities
}
// Called when an applet's window must be restored.
public void paint(Graphics g) {
// redisplay contents of window
}
}
```

Although this skeleton does not do anything, it can be compiled and run. When run, itgenerates the following window when viewed with an applet viewer:

Fig. 1.2



Applet started.

Applet Initialization and Termination :

It is important to understand the order in which the various methods shown in theskeleton are called. When an applet begins, the AWT calls the following methods, in this sequence:

1. init( )
2. start( )
3. paint( )

When an applet is terminated, the following sequence of method calls takes place:

1. stop( )
2. destroy( )

Let’s look more closely at these methods.

init( ) :

The init( ) method is the first method to be called. This is where you should initializevariables. This method is called only once during the run time of your applet.

start( ) :

The start( ) method is called after init( ). It is also called to restart an applet after it hasbeen stopped. Whereas init( ) is called once—the first time an applet is loaded—start( )is called each time an applet's HTML document is displayed onscreen. So, if a userleaves a web page and comes back, the applet resumes execution at start( ).

paint( ) :

The paint( ) method is called each time your applet's output must be redrawn. Thissituation can occur for several reasons. For example, the window in which the applet isrunning may be overwritten by another window and then uncovered. Or the appletwindow may be minimized and then restored. paint( ) is also called when the appletbegins execution. Whatever the cause, whenever the applet must redraw its output,paint( ) is called. The paint( ) method has one parameter of type Graphics. Thisparameter will contain the graphics context, which describes the graphics environmentin which the applet is running. This context is used whenever output to the appletis required.

stop( ):

The stop( ) method is called when a web browser leaves the HTML document containingthe applet—when it goes to another page, for example. When stop( ) is called, theapplet is probably running. You should use stop( ) to suspend threads that don't needto run when the applet is not visible. You can restart them when start( ) is called if theuser returns to the page.

destroy( ) :

The destroy( ) method is called when the environment determines that your appletneeds to be removed completely from memory. At this point, you should freeup any resources the applet may be using. The stop( ) method is always calledbefore destroy( ).

Overriding update( ) :

In some situations, your applet may need to override another method defined by theAWT, called update( ). This method is called when your applet has requested that aportion of its window be redrawn. The default version of update( ) first fills an appletwith the default background color and then calls paint( ). If you fill the backgroundusing a different color in paint( ), the user will experience a flash of the defaultbackground each time update( ) is called—that is, whenever the window is repainted.One way to avoid this problem is to override the update( ) method so that it performsall necessary display activities. Then have paint( ) simply call update( ). Thus, for someapplications, the applet skeleton will override paint( ) and update( ), as shown here:

```
public void update(Graphics g) {
// redisplay your window, here.
}
public void paint(Graphics g) {
update(g);
}
```

For the examples in this book, we will override update( ) only when needed.

Simple Applet Display Methods:

As we've mentioned, applets are displayed in a window and they use the AWT toperform input and output. Although we will examine the methods, procedures, andtechniques necessary to fully handle the AWT windowed environment in subsequentchapters, a few are described here, because we will use them to write sample applets.As we described in Chapter 12, to output a string to an applet, use drawString( ),which is a member of the Graphics class. Typically, it is called from within eitherupdate( ) or paint( ). It has the following general form:
void drawString(String *message*, int *x*, int *y*)

Here, *message* is the string to be output beginning at *x,y.* In a Java window, theupper-left corner is location 0,0. The drawString( ) method will not recognize newlinecharacters. If you want to start a line of text on another line, you must do so manually,specifying the precise X,Y location where you want the line to begin. (As you will see in later chapters, there are techniques that ake this process easy.)To set the background color of an applet's window, use setBackground( ). To set theforeground color (the color in which text is shown, for example), use setForeground( ).

These methods are defined by Component, and they have the following general forms:

void setBackground(Color *newColor*)
void setForeground(Color *newColor*)

Here, *newColor* specifies the new color. The class Color defines the constants shownhere that can be used to specify colors:

Color.black
Color.cyan
Color.gray
Color.lightGray
Color.magenta
Color.pink
Color.white
Color.blue
Color.darkGray
Color.green
Color.orange
Color.red
Color.yellow

For example, this sets the background color to green and the text color to red:

```
setBackground(Color.green);
setForeground(Color.red);
```

A good place to set the foreground and background colors is in the init( ) method.Of course, you can change these colors as often as necessary during the execution ofyour applet. The default foreground color is black. The default background color islight gray.You can obtain the current settings for the background and foreground colors bycalling getBackground( ) and etForeground( ), respectively. They are also definedby Component and are shown here:

```
Color getBackground( )
Color getForeground( )
```

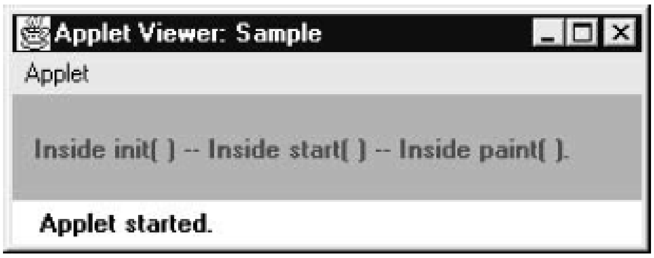
Here is a very simple applet that sets the background color to cyan, the foregroundcolor to red, and displays a message that illustrates the order in which the init( ),start( ), and paint( ) methods are called when an applet starts up:

```
/* A simple applet that sets the foreground and
background colors and outputs a string. */
import java.awt.*;
import java.applet.*;
/*
<applet code="Sample" width=300 height=50>
</applet>
*/
public class Sample extends Applet{
String msg;
// set the foreground and background colors.
public void init() {
setBackground(Color.cyan);
setForeground(Color.red);
msg = "Inside init( ) --";
}
// Initialize the string to be displayed.
public void start() {
msg += " Inside start( ) --";
}
// Display msg in applet window.
public void paint(Graphics g) {
msg += " Inside paint( ).";
g.drawString(msg, 10, 30);
}
}
```

This applet generates the window shown here:

The methods stop( ) and destroy( ) are not overridden, because they are not needed

by this simple applet.



```
Inside init( ) -- Inside start( ) -- Inside paint( ).
```

Applet started.

Requesting Repainting :

As a general rule, an applet writes to its window only when its update( ) or paint( )method is called by the AWT. This raises an interesting question: How can the appletitself cause its window to be updated when its information changes? For example, if anapplet is displaying a moving banner, what mechanism does the applet use to updatethe window each time this banner scrolls? Remember, one of the fundamentalarchitectural constraints imposed on an applet is that it must quickly return control tothe AWT run-time system. It cannot create a loop inside paint( ) that repeatedly scrollsthe banner, for example. This would prevent control from passing back to the AWT.Given this constraint, it may seem that output to your applet's window will be difficultat best. Fortunately, this is not the case. Whenever your applet needs to update theinformation displayed in its window, it simply calls repaint( ).The repaint( ) method is defined by the AWT. It causes the AWT run-timesystem to execute a call to your applet's update( ) method, which, in its defaultimplementation, calls paint( ). Thus, for another part of your applet to output to itswindow, simply store the output and then call repaint( ). The AWT will then executea call to paint( ), which can display the stored information. For example, if part of yourapplet needs to output a string, it can store this string in a String variable and then call
repaint( ). Inside paint( ), you will output the string using drawString( ).The repaint( ) method has four forms. Let's look at each one, in turn. The simplestversion of repaint( ) is shown here:

```
void repaint( )
```

This version causes the entire window to be repainted. The following versionspecifies a region that will be repainted:

void repaint(int *left*, int *top*, int *width*, int *height*)

Here, the coordinates of the upper-left corner of the region are specified by *left* and*top*, and the width and height of the region are passed in *width* and *height.* Thesedimensions are specified in pixels. You save time by specifying a region to repaint.Window updates are costly in terms of time. If you need to update only a small portionof the window, it is more efficient to repaint only that region.Calling repaint( ) is essentially a request that your applet be repainted sometime

soon. However, if your system is slow or busy, update( ) might not be calledimmediately. Multiple requests for repainting that occur within a short time can becollapsed by the AWT in a manner such that update( ) is only called sporadically. Thiscan be a problem in many situations, including animation, in which a consistent updatetime is necessary. One solution to this problem is to use the following forms of repaint( ):

void repaint(long *maxDelay*)
void repaint(long *maxDelay*, int *x*, int *y*, int *width*, int *height*)

Here, *maxDelay* specifies the maximum number of milliseconds that can elapse beforeupdate( ) is called. Beware, though. If the time elapses before update( ) can be called, itisn’t called. There’s no return value or exception thrown, so you must be careful.*It is possible for a method other than paint( ) or update( ) to output to an applet’swindow. To do so, it must obtain a graphics context by calling getGraphics( ) (definedby Component) and then use this context to output to the window. However, for mostapplications, it is better and easier to route window output through paint( ) and to callrepaint( ) when the contents of the window change.*

A Simple Banner Applet :

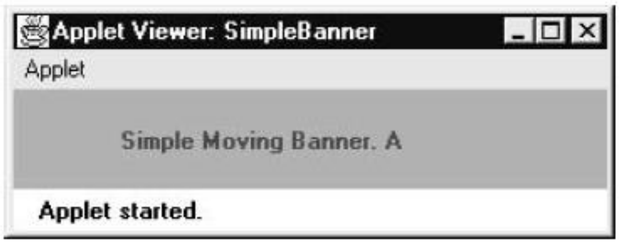
To demonstrate repaint( ), a simple banner applet is developed. This applet scrollsa message, from right to left, across the applet’s window. Since the scrolling of themessage is a repetitive task, it is performed by a separate thread, created by the appletwhen it is initialized. The banner applet is shown here:

```
/* A simple banner applet.
This applet creates a thread that scrolls
the message contained in msg right to left
across the applet's window.
*/
import java.awt.*;
import java.applet.*;
/*
<applet code="SimpleBanner" width=300 height=50>
```

```
</applet>
*/
public class SimpleBanner extends Applet implements Runnable
{
String msg = " A Simple Moving Banner.";
Thread t = null;
int state;
boolean stopFlag;
// Set colors and initialize thread.
public void init() {
setBackground(Color.cyan);
setForeground(Color.red);
}
// Start thread
public void start() {
t = new Thread(this);
stopFlag = false;
t.start();
}
// Entry point for the thread that runs the banner.
public void run() {
char ch;
// Display banner
for( ; ; ) {
try {
repaint();
Thread.sleep(250);
ch = msg.charAt(0);
msg = msg.substring(1, msg.length());
msg += ch;
if(stopFlag)
break;
} catch(InterruptedException e) {}
}
}
// Pause the banner.
public void stop() {
stopFlag = true;
t = null;
}
// Display the banner.
public void paint(Graphics g) {
g.drawString(msg, 50, 30);
}
}
```

Following is sample output:

Fig. 1.4



Let’s take a close look at how this applet operates. First, notice that SimpleBannerextends Applet, as expected, but it also implements Runnable. This is necessary, sincethe applet will be creating a second thread of execution that will be used to scroll thebanner. Inside init( ), the foreground and background colors of the applet are set.After initialization, the AWT run-time system calls start( ) to start the applet running.Inside start( ), a new thread of execution is created and assigned to the Thread variablet. Then, the boolean variable stopFlag, which controls the execution of the applet, is setto false. Next, the thread is started by a call to t.start( ). Remember that t.start( ) calls amethod defined by Thread, which causes run( ) to begin executing. It does not causea call to the version of start( ) defined by Applet. These are two separate methods.Inside run ( ), the characters in the string contained in msg are repeatedly rotated left.Between each rotation, a call to repaint( ) is made. This eventually causes the paint( )method to be called and the current contents of msg is displayed. Between eachiteration, run( ) sleeps for a quarter of a second. The net effect of run( ) is that thecontents of msg is scrolled right to left in a constantly moving display. The stopFlagvariable is checked on each iteration. When it is true, the run( ) method terminates.If a browser is displaying the applet when a new page is viewed, the stop( ) method iscalled, which sets stopFlag to true, causing run ( ) to terminate. This is the mechanism usedto stop the thread when its page is no longer in view. When the applet is brought back intoview, start( ) is once again called, which starts a new thread to execute the banner.

Using the Status Window :

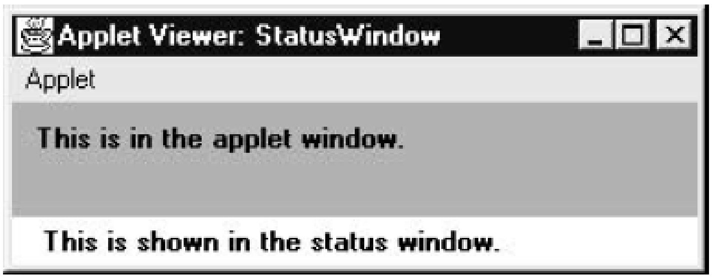
In addition to displaying information in its window, an applet can also output amessage to the status window of the browser or applet viewer on which it is running.To do so, call showStatus( ) with the string that you want displayed. The status windowis a good place to give the user feedback about what is occurring in the applet, suggestoptions, or possibly report some types of errors. The status window also makes anexcellent

debugging aid, because it gives you an easy way to output information aboutyour applet.
The following applet demonstrates showStatus( ):

```
// Using the Status Window.
import java.awt.*;
import java.applet.*;
/*
<applet code="StatusWindow" width=300 height=50>
</applet>
*/
public class StatusWindow extends Applet{
public void init() {
setBackground(Color.cyan);
}
// Display msg in applet window.
public void paint(Graphics g) {
g.drawString("This is in the applet window.", 10, 20);
showStatus("This is shown in the status window.");
}
}
```

Sample output from this program is shown here:

Fig. 1.6



```
This is in the applet window.
```

```
This is shown in the status window.
```

The HTML APPLET Tag :

The APPLET tag is used to start an applet from both an HTML document and from anapplet viewer. (The newer OBJECT tag also works, but this book will use APPLET.)An applet viewer will execute each APPLET tag that it finds in a separate window,while web browsers like Netscape Navigator, Internet Explorer, and HotJava will allowmany applets on a single page. So far, we have been using only a simplified form of theAPPLET tag. Now it is time to take a closer look at it.

The syntax for the standard APPLET tag is shown here. Bracketed items
are optional.

< APPLET
[CODEBASE = *codebaseURL*]
CODE = *appletFile*
[ALT = *alternateText*]
[NAME = *appletInstanceName*]
WIDTH = *pixels* HEIGHT = *pixels*
[ALIGN = *alignment*]
[VSPACE = *pixels*] [HSPACE = *pixels*]
>
[< PARAM NAME = *AttributeName* VALUE = *AttributeValue*>]
[< PARAM NAME = *AttributeName2* VALUE = *AttributeValue*>]
. . .
[*HTML Displayed in the absence of Java*]
</APPLET>

Let's take a look at each part now.

CODEBASE CODEBASE is an optional attribute that specifies the base URL of theapplet code, which is the directory that will be searched for the applet's executableclass file (specified by the CODE tag). The HTML document's URL directory is used asthe CODEBASE if this attribute is not specified. The CODEBASE does not have to beon the host from which the HTML document was read.CODE CODE is a required attribute that gives the name of the file containing yourapplet's compiled .class file. This file is relative to the code base URL of the applet,which is the directory that the HTML file was in or the directory indicated byCODEBASE if set.ALT The ALT tag is an optional attribute used to specify a short text message thatshould be displayed if the browser understands the APPLET tag but can't currentlyrun Java applets. This is distinct from the alternate HTM you provide for browsersthat don't support applets.NAME NAME is an optional attribute used to specify a name for the applet instance.Applets must be named in order for other applets on the same page to find them byname and communicate with them. To obtain an applet by name, use getApplet( ),which is defined by the AppletContext interface.WIDTH AND HEIGHT WIDTH and HEIGHT are required attributes that give thesize (in pixels) of the applet display area.ALIGN ALIGN is an optional attribute that specifies the alignment of the applet.This attribute is treated the same as the HTML IMG tag with these possible values:LEFT, RIGHT, TOP, BOTTOM, MIDDLE, BASELINE, TEXTTOP, ABSMIDDLE,and ABSBOTTOM.VSPACE AND HSPACE These attributes are optional. VSPACE specifies the space,in pixels, above and below the applet. HSPACE specifies the space, in pixels, on eachside of the applet. They're treated the same as the IMG tag's VSPACE and HSPACEattributes.PARAM NAME AND VALUE The PARAM tag allows you to specify appletspecificarguments in an HTML page. Applets access their attributes with thegetParameter( ) method.

HANDLING OLDER BROWSERSSome very old web browsers can’t execute appletsand don’t recognize the APPLET tag. Although these browsers are now nearly extinct(having been replaced by Java-compatible ones), you may need to deal with themoccasionally. The best way to design your HTML page to deal with such browsers is toinclude HTML text and markup within your <applet></applet> tags. If the applet tagsare not recognized by your browser, you will see the alternate markup. If Java isavailable, it will consume all of the markup between the <applet></applet> tags anddisregard the alternate markup.

Here’s the HTML to start an applet called SampleApplet in Java and to display amessage in older browsers:<applet code="SampleApplet" width=200 height=40>

If you were driving a Java powered browser,you'd see &quote;A Sample Applet&quote; e.<p>
</applet>

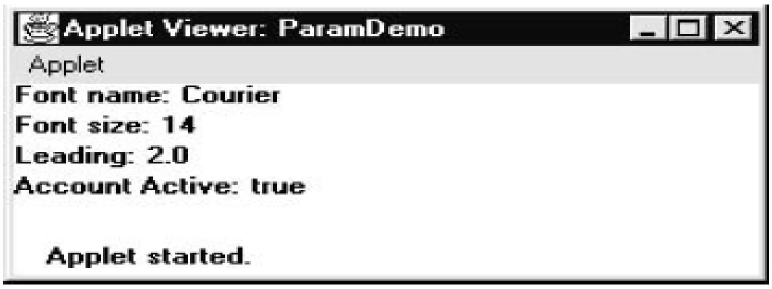
Passing Parameters to Applets :

The APPLET tag in HTML allows you to pass parameters to yourapplet. To retrieve a parameter, use the getParameter( ) method. It returns the value ofthe specified parameter in the form of a String object. Thus, for numeric and Boolean values, you will need to convert their string representations into their internal formats.Here is an example that emonstrates passing parameters:

```
// Use Parameters
import java.awt.*;
import java.applet.*;
/*
<applet code="ParamDemo" width=300 height=80>
<param name=fontName value=Courier>
<param name=fontSize value=14>
<param name=leading value=2>
<param name=accountEnabled value=true>
</applet>
*/
public class ParamDemo extends Applet{
String fontName;
int fontSize;
float leading;
boolean active;
// Initialize the string to be displayed.
public void start() {
String param;
fontName = getParameter("fontName");
if(fontName == null)
fontName = "Not Found";
param = getParameter("fontSize");
try {
if(param != null) // if not found
fontSize = Integer.parseInt(param);
else
fontSize = 0;
} catch(NumberFormatException e) {
fontSize = -1;
}
param = getParameter("leading");
try {
if(param != null) // if not found
leading = Float.valueOf(param).floatValue();
else
leading = 0;
} catch(NumberFormatException e) {
leading = -1;
}
param = getParameter("accountEnabled");
if(param != null)
active = Boolean.valueOf(param).booleanValue();
}
// Display parameters.
public void paint(Graphics g) {
g.drawString("Font name: " + fontName, 0, 10);
g.drawString("Font size: " + fontSize, 0, 26);
g.drawString("Leading: " + leading, 0, 42);
g.drawString("Account Active: " + active, 0, 58);
}
}
```

Sample output from this program is shown here:

Fig. 1.7



```
Font name: Courier
Font size: 14
Leading: 2.0
Account Active: true
```

As the program shows, you should test the return values from getParameter( ). If aparameter isn’t available, getParameter( ) will return null. Also, conversions to numerictypes must be attempted in a try statement that catches NumberFormatException.Uncaught exceptions should never occur within an applet.

Improving the Banner Applet :

It is possible to use a parameter to enhance the banner applet shown earlier. In theprevious version, the message being scrolled was hard-coded into the applet. However,passing the message as a parameter allows the banner applet to display a differentmessage each time it is executed. This improved version is shown here. Notice that theAPPLET tag at the top of the file now specifies a parameter called message that is linkedto a quoted string.

```
// A parameterized banner
import java.awt.*;
import java.applet.*;
/*
<applet code="ParamBanner" width=300 height=50>
<param name=message value="Java makes the Web move!">
</applet>
*/
public class ParamBanner extends Applet implements Runnable
{
String msg;
Thread t = null;
int state;
boolean stopFlag;
// Set colors and initialize thread.
public void init() {
setBackground(Color.cyan);
setForeground(Color.red);
}
// Start thread
public void start() {
msg = getParameter("message");
if(msg == null) msg = "Message not found.";
msg = " " + msg;
t = new Thread(this);
stopFlag = false;
t.start();
}
// Entry point for the thread that runs the banner.
public void run() {

char ch;
// Display banner
for( ; ; ) {
try {
repaint();
Thread.sleep(250);
ch = msg.charAt(0);
msg = msg.substring(1, msg.length());
msg += ch;
if(stopFlag)
break;
} catch(InterruptedException e) {}
}
}
// Pause the banner.
public void stop() {
stopFlag = true;
t = null;
}
// Display the banner.
public void paint(Graphics g) {
g.drawString(msg, 50, 30);
}
}
```

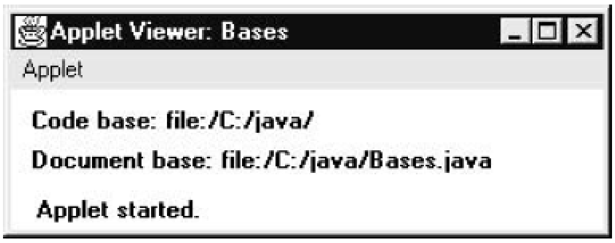
getDocumentBase( ) and getCodeBase( ) :

Often, you will create applets that will need to explicitly load media and text. Java willallow the applet to load data from the directory holding the HTML file that started theapplet (the *document base*) and the directory from which the applet's class file wasloaded (the *code base*). These directories are returned as URL objects by getDocumentBase( ) and getCodeBase( ). They can be concatenatedwith a string that names the file you want to load. To actually load another file, youwill use the showDocument( ) method defined by theAppletContext interface,discussed in the next section.The following applet illustrates these methods:

```
// Display code and document bases.
import java.awt.*;
import java.applet.*;
import java.net.*;
/*
<applet code="Bases" width=300 height=50>
</applet>
*/
public class Bases extends Applet{
// Display code and document bases.
public void paint(Graphics g) {
String msg;
URL url = getCodeBase(); // get code base
msg = "Code base: " + url.toString();
g.drawString(msg, 10, 20);
url = getDocumentBase(); // get document base
msg = "Document base: " + url.toString();
g.drawString(msg, 10, 40);
}
}
```

Sample output from this program is shown here:

Fig. 1.8



```
Code base: file:/C:/java/
Document base: file:/C:/java/Bases.java

Applet started.
```

AppletContext and showDocument( ):

One application of Java is to use active images and animation to provide a graphicalmeans of navigating the Web that is more interesting than the underlined bluewords used by hypertext. To allow your applet to transfer control to another URL,you must use the showDocument( ) method defined by the AppletContext interface.AppletContext is an interface that lets you get nformation from the applet’s execution environment. Thecontext of the currently executing applet is obtained by a call to the getAppletContext( )method defined by Applet.Within an applet, once you have obtained the applet's context, you can bringanother document into view by calling showDocument( ). This method has noreturn value and throws no exception if it fails, so use it carefully. There are twoshowDocument( ) methods. The method showDocument(URL) displays the document

Tabel 1.9

Method	**Description**
Applet getApplet (String *appletName*)	Returns the applet specified by *appletName* if it iswithin the current applet context.Otherwise, nullis returned.
Enumeration getApplets( )	Returns an enumeration that contains all of theapplets within the current applet context.
AudioClip getAudioClip(URL *url*)	Returns an AudioClip object that encapsulates theaudio clip found at the location specified by *url.*
Image getImage(URL *url*)	Returns an Image object that encapsulates the imagefound at the location specified by *url.*
InputStream getStream (String *key*)	Returns the stream linked to *key*. Keys are linked tostreams by using the setStream( ) method. A nullreference is returned if no stream is linked to *key*.
Iterator getStreamKeys( )	Returns an iterator for the keys associated with theinvoking object. The keys are linked to streams. See
getStream( ) and setStream( ).	
void setStream(String *key*, InputStream *strm*)	Links the stream specified by *strm* to the key passedin *key*. The *key* is deleted from the invoking objectif*strm* is null.
void showDocument(URL *url*)	Brings the document at the URL specified by *url*into view. This method may not be supported byapplet viewers.
void showDocument(URL *url*, String *where*)	Brings the document at the URL specified by *url*into view. This method may not be

	supported byapplet viewers. The placement of the documentis specified by *where* as described in the text.
void showStatus(String *str*)	Displays *str* in the status window.

The Abstract Methods Defined by the AppletContext Interfaceat the specified URL. The method showDocument(URL, where) displays the specifieddocument at the specified location within the browser window. Valid arguments for*where* are “_self” (show in current frame), “_parent” (show in parent frame), “_top”(show in topmost frame), and “_blank” (show in new browser window). You can alsospecify a name, which causes the document to be shown in a new browser window bythat name.The following applet demonstrates AppletContext and showDocument( ).Upon execution, it obtains the current applet context and uses that context totransfer control to a file called Test.html. This file must be in the same directoryas the applet. Test.html can contain any valid hypertext that you like.

```
/* Using an applet context, getCodeBase(),
and showDocument() to display an HTML file.
*/
import java.awt.*;
import java.applet.*;
import java.net.*;
/*
<applet code="ACDemo" width=300 height=50>
</applet>
*/
public class ACDemo extends Applet{
public void start() {
AppletContext ac = getAppletContext();
URL url = getCodeBase(); // get url of this applet
try {
ac.showDocument(new URL(url+"Test.html"));
} catch(MalformedURLException e) {
showStatus("URL not found");
}
}
}
```

The AudioClip Interface :

The AudioClip interface defines these methods: play( ) (play a clip from thebeginning), stop( ) (stop playing the clip), and loop( ) (play the loop continuously).After you have loaded an audio clip using getAudioClip( ), you can use these methodsto play it.

The AppletStub Interface :

The AppletStub interface provides the means by which an applet and the browser (orapplet viewer) communicate. Your code will not typically implement this interface.

Outputting to the Console :

Although output to an applet's window must be accomplished through AWTmethods, such as drawString( ), it is still possible to use console output in yourapplet—especially for debugging purposes. In an applet, when you call a methodsuch as System.out.println( ), the output is not sent to your applet's window. Instead,it appears either in the console session in which you launched the applet viewer or inthe Java console that is available in some browsers. Use of console output for purposesother than debugging is discouraged, since it violates the design principles of thegraphical interface most users will expect.

1.12 MANAGING FILES AND STREAMS

This chapter explores java.io, which provides support for I/O operations. InPrevious chapter , we presented an overview of Java's I/O system. Here, we willexamine the Java I/O system in greater detail.As all programmers learn early on, most programs cannot accomplish their goalswithout accessing external data. Data is retrieved from an *input* source. The results ofa program are sent to an *output* destination. In Java, these sources or destinations aredefined very broadly. For example, a network connection, memory buffer, or disk filecan be manipulated by the Java I/O classes. Although physically different, thesedevices are all handled by the same abstraction: the *stream.* A stream, as explained in is a logical entity that either produces or consumes information. A streamis linked to a physical device by the Java I/O system. All streams behave in the samemanner, even if the actual physical devices they are linked to differ.

The Java I/O Classes and Interfaces :

The I/O classes defined by java.io are listed here:

BufferedInputStream
BufferedOutputStream
BufferedReader
BufferedWriter
FileWriter
FilterInputStream
FilterOutputStream
FilterReader
PipedInputStream
PipedOutputStream
PipedReader
PipedWriter

ByteArrayInputStream
ByteArrayOutputStream
CharArrayReader
CharArrayWriter
DataInputStream
DataOutputStream
File
FileDescriptor
SerializablePermission
FileInputStream
FileOutputStream
FilePermission
FileReader
FilterWriter
InputStream
InputStreamReader
LineNumberReader
ObjectInputStream
ObjectInputStream.
ObjectOutputStream
ObjectOutputStream.
ObjectStreamClass
ObjectStreamField
OutputStream
OutputStreamWriter
PrintStream
PrintWriter
PushbackInputStream
PushbackReader
RandomAccessFile
GetField Reader
SequenceInputStream
PutField
StreamTokenizer
StringReader
StringWriter
Writer

The ObjectInputStream.GetField and ObjectOutputStream.PutField inner classeswere added by Java 2. The java.io package also contains two classes that were deprecatedby Java 2 and are not shown in the preceding table: LineNumberInputStream andStringBufferInputStream. These classes should not be used for new code.The following interfaces are defined by java.io:

DataInput
DataOutput
Externalizable
FileFilter
FilenameFilter
ObjectInput
ObjectInputValidation
ObjectOutput
ObjectStreamConstants
Serializable

The FileFilter interface was added by Java 2.

As you can see, there are many classes and interfaces in the java.io package. Theseinclude byte and character streams, and object serialization (the storage and retrieval ofobjects).

File :

Although most of the classes defined by java.io operate on streams, the File class doesnot. It deals directly with files and the file system. That is, the File class does notspecify how information is retrieved from or stored in files; it describes the propertiesof a file itself. A File object is used to obtain or manipulate the information associatedwith a disk file, such as the permissions, time, date, and directory path, and to navigatesubdirectory hierarchies.Files are a primary source and destination for data within many programs.Although there are severe restrictions on their use within applets for security reasons,files are still a central resource for storing persistent and shared information. Adirectory in Java is treated simply as a File with one additional property—a list offilenames that can be examined by the list( ) method.

The following constructors can be used to create File objects:

File(String *directoryPath*)
File(String *directoryPath,* String *filename*)
File(File *dirObj*, String *filename*)
File(URI *uriObj*)

Here, *directoryPath* is the path name of the file, *filename* is the name of the file, *dirObj* is aFile object that specifies a directory, and *uriObj* is a URI object that describes a file. Thefourth constructor was added by Java 2, version 1.4.The following example creates three files: f1, f2, and f3. The first File object isconstructed with a directory path as the only argument. The second includes twoarguments—the path and the filename. The third includes the file path assigned to f1and a filename; f3 refers to the same file as f2.

```
File f1 = new File("/");
File f2 = new File("/","autoexec.bat");
File f3 = new File(f1,"autoexec.bat");
```

Java does the right thing with path separators between UNIX and Windowsconventions. If you use a forward slash (/) on a Windows version of Java, the pathwill still resolve correctly. Remember, if you are using the Windows conventionof a backslash character (\), you will need to use its escape sequence (\\) within astring. The Java convention is to use the UNIX- and URL-style forward slash for path seprators.

File defines many methods that obtain the standard properties of a File object. Forexample, getName( ) returns the name of the file, getParent( ) returns the name of theparent directory, and exists( ) returns true if the file exists, false if it does not. The File class,however, is not symmetrical. By this, we mean that there are many methods that allow youto *examine* the properties of a simple file object, but no corresponding function exists tochange those attributes. The following example demonstrates several of the File methods:

```
// Demonstrate File.
import java.io.File;
class FileDemo {
static void p(String s) {
System.out.println(s);
}
public static void main(String args[]) {
File f1 = new File("/java/COPYRIGHT");
p("File Name: " + f1.getName());
p("Path: " + f1.getPath());
p("Abs Path: " + f1.getAbsolutePath());
p("Parent: " + f1.getParent());
p(f1.exists() ? "exists" : "does not exist");
p(f1.canWrite() ? "is writeable" : "is not writeable");
p(f1.canRead() ? "is readable" : "is not readable");
p("is " + (f1.isDirectory() ? "" : "not" + " a directory"));
p(f1.isFile() ? "is normal file" : "might be a named pipe");
p(f1.isAbsolute() ? "is absolute" : "is not absolute");
p("File last modified: " + f1.lastModified());
p("File size: " + f1.length() + " Bytes");
}
}
```

When you run this program, you will see something similar to the following:

```
File Name: COPYRIGHT
Path: /java/COPYRIGHT
Abs Path: /java/COPYRIGHT
Parent: /java
exists
is writeable
is readable
is not a directory
is normal file
is absolute
File last modified: 812465204000
File size: 695 Bytes
```

Most of the File methods are self-explanatory. isFile( ) and isAbsolute( ) are not.isFile( ) returns true if called on a file and false if called on a directory. Also, isFile( )returns false for some special files, such as device drivers and named pipes, so thismethod can be used to make sure the file will behave as a file. The isAbsolute( )method returns true if the file has an absolute path and false if its path is relative.File also includes two useful utility methods. The first is renameTo( ), shown here:boolean renameTo(File *newName*)Here, the filename specified by *newName* becomes the new name of the invoking Fileobject. It will return true upon success and false if the file cannot be renamed (if youeither attempt to rename a file so that it moves from one directory to another or use anexisting filename, for example).The second utility method is delete( ), which deletes the disk file represented by thepath of the invoking File object. It is shown here:

boolean delete( ) :

You can also use delete( ) to delete a directory if the directory is empty. delete( )returns true if it deletes the file and false if the file cannot be removed.

Here are some other File methods that you will find helpful. (They were addedby Java 2.)

Tabel 1.10

Method	Description
void deleteOnExit( )	Removes the file associated with theinvoking object when the Java VirtualMachine terminates.
boolean isHidden( )	Returns true if the invoking file ishidden. Returns false otherwise.
boolean setLastModified (long *millisec*)	Sets the time stamp on the invokingfile to that specified by *millisec*, whichis the number of milliseconds fromJanuary 1, 1970, CoordinatedUniversal Time (UTC).boolean setReadOnly( ) Sets the invoking file to read-only.Also, because File supports the Comparable interface, the method compareTo( ) isalso supported.

Directories :

A directory is a File that contains a list of other files and directories. When you create a Fileobject and it is a directory, the isDirectory( ) method will return true. In this case, you can call list( ) on that object to extract the list of other files and directories inside .

It has two forms. The first is shown here:

String[ ] list( ) The list of files is returned in an array of String objects.

The program shown here illustrates how to use list( ) to examine the contents ofa directory:

```
// Using directories.
import java.io.File;
class DirList {
public static void main(String args[]) {
String dirname = "/java";
File f1 = new File(dirname);
if (f1.isDirectory()) {
```

```
System.out.println("Directory of " + dirname);
String s[] = f1.list();
for (int i=0; i < s.length; i++) {
File f = new File(dirname + "/" + s[i]);
if (f.isDirectory()) {
System.out.println(s[i] + " is a directory");
} else {
System.out.println(s[i] + " is a file");
}
}
} else {
System.out.println(dirname + " is not a directory");
}
}}
```

Here is sample output from the program

```
Directory of /java
bin is a directory
lib is a directory
demo is a directory
COPYRIGHT is a file
README is a file
index.html is a file
include is a directory
src.zip is a file
.hotjava is a directory
src is a directory
```

Using FilenameFilter :

You will often want to limit the number of files returned by the list( ) method toinclude only those files that match a certain filename pattern, or *filter.* To do this, youmust use a second form of list( ), shown here:

String[ ] list(FilenameFilter *FFObj*)

In this form, *FFObj* is an object of a class that implements the FilenameFilter interface.

FilenameFilter defines only a single method, accept( ), which is called once for eachfile in a list. Its general form is given here:

boolean accept(File *directory*, String *filename*)

The accept( ) method returns true for files in the directory specified by *directory* thatshould be included in the list (that is, those that match the *filename* argument), andreturns false for those files that should be excluded.

The OnlyExt class, shown next, implements FilenameFilter. It will be used to modifythe preceding program so that it restricts the visibility of the filenames returned by list( )to files with names that end in the file extension specified when the object is constructed.

```
import java.io.*;
public class OnlyExt implements FilenameFilter {
String ext;
public OnlyExt(String ext) {
this.ext = "." + ext;
}
public boolean accept(File dir, String name) {
return name.endsWith(ext);
}
}
```

The modified directory listing program is shown here. Now it will only display files

that use the .html extension.

```
// Directory of .HTML files.
import java.io.*;
class DirListOnly {
public static void main(String args[]) {
String dirname = "/java";
File f1 = new File(dirname);
FilenameFilter only = new OnlyExt("html");
String s[] = f1.list(only);
for (int i=0; i < s.length; i++) {
System.out.println(s[i]);
}
}
}
```

The listFiles( ) Alternative:

Java 2 added a variation to the list( ) method, called listFiles( ), which you might finduseful. The signatures for listFiles( ) are shown here:

File[ ] listFiles( )
File[ ] listFiles(FilenameFilter *FFObj*)
File[ ] listFiles(FileFilter *FObj*)

These methods return the file list as an array of File objects instead of strings. The firstmethod returns all files, and the second returns those files that satisfy the

specifiedFilenameFilter. Aside from returning an array of File objects, these two versions oflistFiles( ) work like their equivalent list( ) methods.The third version of listFiles( ) returns those files with path names that satisfy thespecified FileFilter. FileFilter defines only a single method, accept( ), which is called once for each file in a list. Its general form is given here:
boolean accept(File *path*)

The accept( ) method returns true for files that should be included in the list (that is,those that match the *path* argument), and false for those that should be excluded.

Creating Directories :

Another two useful File utility methods are mkdir( ) and mkdirs( ). The mkdir( )method creates a directory, returning true on success and false on failure. Failureindicates that the path specified in the File object already exists, or that the directorycannot be created because the entire path does not exist yet. To create a directory forwhich no path exists, use the mkdirs( ) method. It creates both a directory and all theparents of the directory.

The Stream Classes :

Java's stream-based I/O is built upon four abstract classes: InputStream, OutputStream, Reader, and Writer. They are used to create several concrete stream subclasses.Although your programs perform their I/O operations through concrete subclasses, the top-level classes define the basic functionality common to all stream classes.InputStream and OutputStream are designed for byte streams. Reader and Writerare designed for character streams. The byte stream classes and the character streamclasses form separate hierarchies. In general, you should use the character streamclasses when working with characters or strings, and use the byte stream classes when working with bytes or other binary obj ects.In the remainder of this chapter, both the byte- and character-oriented streamsare examined.

The Byte Streams :

The byte stream classes provide a rich environment for handling byte-oriented I/O. Abyte stream can be used with any type of object, including binary data. This versatilitymakes byte streams important to many types of programs. Since the byte stream classes are topped by InputStream and OutputStream, our discussion will begin with them.

InputStream :

InputStream is an abstract class that defines Java's model of streaming byte input. Allof the methods in this class will throw an IOException on error conditions. Table shows the methods in InputStream.

Tabel 1.11

Method	**Description**
int available( )	Returns the number of bytes of input currentlyavailable for reading.
void close( )	Closes the input source. Further read attemptswill generate an IOException.
void mark(int *numBytes*)	Places a mark at the current point in the input stream that will remain valid until *numBytes*bytes are read.
boolean markSupported( )	Returns true if mark( )/reset( ) are supported by the invoking stream.
int read( )	Returns an integer representation of the nextavailable byte of input. –1 is returned when theend of the file is encountered.
int read(byte *buffer*[ ])	Attempts to read up to *buffer*.*length* bytes into*buffer* and returns the actual number of bytesthat were successfully read. –1 is returned when the end of the file is encountered.
int read(byte *buffer*[ ], int *offset*,	
int *numBytes*)	Attempts to read up to *numBytes* bytes into
buffer starting at *buffer*[*offset*],	returning the number of bytes successfully read. –1 is returned when the end of the file isencountered.

OutputStream:

OutputStream is an abstract class that defines streaming byte output. All of themethods in this class return a void value and throw an IOException in the case oferrors. Table 1.12 shows the methods in OutputStream.

Table 1.12

Method	**Description**
void reset( )	Resets the input pointer to the previouslyset mark.
long skip(long *numBytes*)	Ignores (that is, skips) *numBytes* bytes of input,returning the number of bytes actually ignored.

Method	**Description**
void close( )	Closes the output stream. Further writeattempts will generate an IOException.
void flush( )	Finalizes the output state so that anybuffers are cleared. That is, it flushes theoutput buffers.
void write(int *b*)	Writes a single byte to an output stream.Note that the parameter is an int, whichallows you to call write ( ) with expressionswithout having to cast them back to byte.
void write(byte *buffer*[ ])	Writes a complete array of bytes to anoutput stream.
void write(byte *buffer*[ ], int *offset*, int *numBytes*)	Writes a subrange of *numBytes* bytes fromthe array *buffer*, beginning at *buffer*[*offset*].

Most of the methods described in Tables 1 and 2 are implemented by thesubclasses of InputStream and OutputStream. The mark( ) and reset( ) methods areexceptions; notice their use or lack thereof by each subclass in the discussions that follow.

FileInputStream:

The FileInputStream class creates an InputStream that you can use to read bytes froma file. Its two most common constructors are shown here:

FileInputStream(String *filepath*)
FileInputStream(File *fileObj*)

Either can throw a FileNotFoundException. Here, *filepath* is the full path name of a file,and *fileObj* is a File object that describes the file.The following example creates two ileInputStreams that use the same disk fileand each of the two constructors:

```
FileInputStream f0 = new FileInputStream("/autoexec.bat")
File f = new File("/autoexec.bat");
FileInputStream f1 = new FileInputStream(f);
```

Although the first constructor is probably more commonly used, the second allowsus to closely examine the file using the File methods, before we attach it to an inputstream. When a FileInputStream is created, it is also opened for reading.FileInputStream overrides six of the methods in the abstract class InputStream. Themark( ) and reset( ) methods are not overridden, and any attempt to use reset( ) on aFileInputStream will generate an IOException.The next example shows how to read a single byte, an array of bytes, and asubrange array of bytes. It also illustrates how to use available ( ) to determine thenumber of bytes remaining, and how to use the skip( ) method to skip over unwantedbytes. The program reads its own source file, which must be in the current directory.

```
// Demonstrate FileInputStream.
import java.io.*;
class FileInputStreamDemo {
public static void main(String args[]) throws Exception {
int size;
InputStream f =
new FileInputStream("FileInputStreamDemo.java");

System.out.println("Total Available Bytes: " +
(size = f.available()));
int n = size/40;
System.out.println("First " + n +
" bytes of the file one read() at a time");
for (int i=0; i < n; i++) {
System.out.print((char) f.read());
}
System.out.println("\nStill Available: " + f.available());
System.out.println("Reading the next " + n +
" with one read(b[])");
byte b[] = new byte[n];
if (f.read(b) != n) {
System.err.println("couldn't read " + n + " bytes.");
}
System.out.println(new String(b, 0, n));
System.out.println("\nStill Available: " + (size = f.available()));
System.out.println("Skipping half of remaining bytes with skip()");
f.skip(size/2);
System.out.println("Still Available: " + f.available());
System.out.println("Reading " + n/2 + " into the end of array");
if (f.read(b, n/2, n/2) != n/2) {
System.err.println("couldn't read " + n/2 + " bytes.");
}
System.out.println(new String(b, 0, b.length));
System.out.println("\nStill Available: " + f.available());
f.close();
}
}
```

Here is the output produced by this program:

```
Total Available Bytes: 1433
First 35 bytes of the file one read() at a time

// Demonstrate FileInputStream.
im
Still Available: 1398
Reading the next 35 with one read(b[])
port java.io.*;
class FileInputS
Still Available: 1363
Skipping half of remaining bytes with skip()
Still Available: 682
Reading 17 into the end of array
port java.io.*;
read(b) != n) {
Still Available: 665
```

This somewhat contrived example demonstrates how to read three ways, to skip input,and to inspect the amount of data available on a stream.Java 2, version 1.4 added the getChannel( ) method to FileInputStream. Thismethod returns a channel connected to the FileInputStream object. Channels are usedby the new I/O classes contained in java.nio.

FileOutputStream:

FileOutputStream creates an OutputStream that you can use to write bytes to a file. Itsmost commonly used constructors are shown here:

FileOutputStream(String *filePath*)

FileOutputStream(File *fileObj*)

FileOutputStream(String *filePath*, boolean *append*)

FileOutputStream(File *fileObj*, boolean *append*)

They can throw a FileNotFoundException or a SecurityException. Here, *filePath* is the full path name of a file, and *fileObj* is a File object that describes the file. If *append* is true, the file is opened in append mode. The fourth constructor was added by Java 2, version 1.4. Creation of a FileOutputStream is not dependent on the file already existing.FileOutputStream will create the file before opening it for output when you create the object. In the case where you attempt to open a read-only file, an IOException will be thrown.

The following example creates a sample buffer of bytes by first making a String and then using the getBytes( ) method to extract the byte array equivalent. It then creates three files. The first, file1.txt, will contain every other byte from the sample. The second, file2.txt, will contain the entire set of bytes. The third and last, file3.txt,will contain only the last quarter. Unlike the FileInputStream methods, all of the FileOutputStream methods have a return type of void. In the case of an error, these methods will throw an IOException.

```
// Demonstrate FileOutputStream.
import java.io.*;
class FileOutputStreamDemo {
public static void main(String args[]) throws Exception {
String source = "Now is the time for all good men\n"
+ " to come to the aid of their country\n"
+ " and pay their due taxes.";
byte buf[] = source.getBytes();
OutputStream f0 = new FileOutputStream("file1.txt");
for (int i=0; i < buf.length; i += 2) {
f0.write(buf[i]);
}
f0.close();
OutputStream f1 = new FileOutputStream("file2.txt");
f1.write(buf);
f1.close();
OutputStream f2 = new FileOutputStream("file3.txt");
f2.write(buf,buf.length-buf.length/4,buf.length/4);
f2.close();
}}
```

Here are the contents of each file after running this program. First, file1.txt:

```
Nwi h iefralgo e
t oet h i ftercuty n a hi u ae.
```

Next, file2.txt:

```
Now is the time for all good men
to come to the aid of their country
and pay their due taxes.
```

Finally, file3.txt:

```
nd pay their due taxes.
```

Java 2, version 1.4 added the getChannel( ) method to FileOutputStream. This method returns a channel connected to the FileOutputStream object. Channels are used by the new I/O classes contained in java.nio.

ByteArrayInputStream:

ByteArrayInputStream is an implementation of an input stream that uses a byte arrayas the source. This class has two constructors, each of which requires a byte array toprovide the data source:

ByteArrayInputStream(byte *array*[ ])
ByteArrayInputStream(byte *array*[ ], int *start*, int *numBytes*)

Here, *array* is the input source. The second constructor creates an InputStream from

A subset of your byte array that begins with the character at the index specified by *start* and is *numBytes* long. The following example creates a pair of ByteArrayInputStreams, initializingthem with the byte representation of the alphabet:

```
// Demonstrate ByteArrayInputStream.
import java.io.*;
class ByteArrayInputStreamDemo {
public static void main(String args[]) throws IOException {
String tmp = "abcdefghijklmnopqrstuvwxyz";
byte b[] = tmp.getBytes();
ByteArrayInputStream input1 = new ByteArrayInputStream(b);
ByteArrayInputStream input2 = new ByteArrayInputStream(b, 0,3);
}
}
```

The input1 object contains the entire lowercase alphabet, while input2 contains onlythe first three letters. A ByteArrayInputStream implements both mark( ) and reset( ). However, if mark( ) has not been called, then reset( ) sets the stream pointer to the start of the stream—which in this case is the start of the byte array passed to the constructor. The next example shows how to use the reset( ) method to read the same input twice. In this case, we read and print the letters “abc” once in lowercase and then again in uppercase.

```
import java.io.*;
class ByteArrayInputStreamReset {
public static void main(String args[]) throws IOException {
String tmp = "abc";
byte b[] = tmp.getBytes();

ByteArrayInputStream in = new ByteArrayInputStream(b);
for (int i=0; i<2; i++) {
int c;
while ((c = in.read()) != -1) {
if (i == 0) {
System.out.print((char) c);
} else {
System.out.print(Character.toUpperCase((char) c));
}
}
System.out.println();
in.reset();
}
}
}
```

This example first reads each character from the stream and prints it as is, in lowercase.It then resets the stream and begins reading again, this time converting each characterto uppercase before printing. Here’s the output:

```
abc
ABC
```

ByteArrayOutputStream:

ByteArrayOutputStream is an implementation of an output stream that uses a bytearray as the destination. ByteArrayOutputStream has two constructors, shown here:

ByteArrayOutputStream( )
ByteArrayOutputStream(int *numBytes*)

In the first form, a buffer of 32 bytes is created. In the second, a buffer is created with a sizeequal to that specified by *numBytes.* The buffer is held in the protected buf field of ByteArrayOutputStream. The buffer size will be increased automatically, if needed. The number of bytes held by the buffer is contained in the protected count field of ByteArrayOutputStream.

The following example demonstrates ByteArrayOutputStream:

```
// Demonstrate ByteArrayOutputStream.
import java.io.*;
class ByteArrayOutputStreamDemo {
public static void main(String args[]) throws IOException {
ByteArrayOutputStream f = new ByteArrayOutputStream();
String s = "This should end up in the array";
byte buf[] = s.getBytes();
f.write(buf);
System.out.println("Buffer as a string");
System.out.println(f.toString());
System.out.println("Into array");
byte b[] = f.toByteArray();
for (int i=0; i<b.length; i++) {
System.out.print((char) b[i]);
}
System.out.println("\nTo an OutputStream()");
OutputStream f2 = new FileOutputStream("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());
}
}
```

When you run the program, you will create the following output. Notice how after the call to reset( ), the three *X'*s end up at the beginning.

```
Buffer as a string
This should end up in the array
Into array
This should end up in the array
To an OutputStream()
Doing a reset
XXX
```

This example uses the writeTo( ) convenience method to write the contents of f totest.txt. Examining the contents of the test.txt file created in the preceding exampleshows the result we expected:This should end up in the array

Filtered Byte Streams :

Filtered streams are simply wrappers around underlying input or output streams thattransparently provide some extended level of functionality. These streams are typicallyaccessed by methods that are expecting a generic stream, which is a superclass of thefiltered streams. Typical extensions are buffering, character translation, and raw datatranslation. The filtered byte streams are FilterInputStream and FilterOutputStream.Their constructors are shown here:

FilterOutputStream(OutputStream *os*)
FilterInputStream(InputStream *is*)

The methods provided in these classes are identical to those in InputStream andOutputStream.

Buffered Byte Streams :

For the byte-oriented streams, a *buffered stream* extends a filtered stream class by attaching a memory buffer to the I/O streams. This buffer allows Java to do I/O operations on more than a byte at a time, hence increasing performance. Because the buffer is available, skipping, marking, and resetting of the stream becomes possible. The buffered byte stream classes areBufferedInputStream and BufferedOutputStream. PushbackInputStream also implements a buffered stream.

BufferedInputStream :

Buffering I/O is a very common performance optimization. Java's BufferedInputStream class allows you to "wrap" any InputStream into a buffered stream and achieve this performance improvement.

BufferedInputStream has two constructors:

BufferedInputStream(InputStream *inputStream*)
BufferedInputStream(InputStream *inputStream*, int *bufSize*)

The first form creates a buffered stream using a default buffer size. In the second, thesize of the buffer is passed in *bufSize.* Use of sizes that are multiples of memory page,disk block, and so on can have a significant positive impact on performance. This is,however, implementation-dependent. An optimal buffer size is generally dependent onthe host operating system, the amount of memory available, and how the machine isconfigured. To make good use of buffering doesn't necessarily require quite this degree of sophistication. A good guess for a size is around 8,192 bytes, and attaching even a rather small buffer to an I/O stream is always a good idea. That way, the low-level system can read blocks of data from the disk or network and store the results in your buffer. Thus, even if you are reading the data a byte at a time out of the InputStream, you will be manipulating fast memory over 99.9 percent of the time.

Buffering an input stream also provides the foundation required to support movingbackward in the stream of the available buffer. Beyond the read( ) and skip( ) methods implemented in any InputStream, BufferedInputStream also supports the mark( ) and reset( ) methods. This support is reflected by BufferedInputStream.markSupported( )returning true.

The following example contrives a situation where we can use mark( ) to rememberwhere we are in an input stream and later use reset( ) to get back there. This example is parsing a stream for the HTML entity reference for the copyright symbol. Such areference begins with an ampersand (&) and ends with a semicolon (;) without anyintervening whitespace. The sample input has two ampersands to show the case where the reset( ) happens and where it does not.

```
// Use buffered input.
import java.io.*;
class BufferedInputStreamDemo {
public static void main(String args[]) throws IOException {
String s = "This is a &copy; copyright symbol " +
"but this is &copy not.\n";
byte buf[] = s.getBytes();
ByteArrayInputStream in = new ByteArrayInputStream(buf);
BufferedInputStream f = new BufferedInputStream(in);
int c;
boolean marked = false;
while ((c = f.read()) != -1) {
switch(c) {
case '&':
if (!marked) {
f.mark(32);
marked = true;
} else {
marked = false;
}
break;
case ';':
if (marked) {
marked = false;
System.out.print("(c)");
} else
System.out.print((char) c);
break;
case ' ':
if (marked) {
marked = false;
f.reset();
System.out.print("&");
} else
System.out.print((char) c);
break;
default:
if (!marked)
System.out.print((char) c);
break;
}
}
}
}
```

Notice that this example uses mark(32), which preserves the mark for the next 32 bytesread (which is enough for all entity references). Here is the output produced by thisprogram:

```
This is a (c) copyright symbol but this is &copy not.
```

Use of mark( ) is restricted to access within the buffer. This means that you can onlyspecify a parameter to mark( ) that is smaller than the buffer size of the stream.

BufferedOutputStream

A BufferedOutputStream is similar to any OutputStream with the exception of anadded flush( ) method that is used to ensure that data buffers are physically written tothe actual output device. Since the point of a BufferedOutputStream is to improveperformance by reducing the number of times the system actually writes data, you mayneed to call flush( ) to cause any data that is in the buffer to get written.Unlike buffered input, buffering output does not provide additional functionality.Buffers for output in Java are there to increase performance. Here are the two available constructors:

BufferedOutputStream(OutputStream *outputStream*)
BufferedOutputStream(OutputStream *outputStream*, int *bufSize*)

The first form creates a buffered stream using a buffer of 512 bytes. In the second form, the size of the buffer is passed in *bufSize.*

PushbackInputStream:

One of the novel uses of buffering is the implementation of pushback. *Pushback* is used on an input stream to allow a byte to be read and then returned (that is, “pushedback”) to the stream. The PushbackInputStream class implements this idea. It provides a mechanism to “peek” at what is coming from an input stream without disrupting it.

PushbackInputStream has the following constructors:

PushbackInputStream(InputStream *inputStream*)
PushbackInputStream(InputStream *inputStream,* int *numBytes*)

The first form creates a stream object that allows one byte to be returned to the inputstream. The second form creates a stream that has a pushback buffer that is *numBytes* long. This allows multiple bytes to be returned to the input stream.Beyond the familiar methods of InputStream, PushbackInputStream provides

unread( ), shown here:

void unread(int *ch*)
void unread(byte *buffer*[ ])
void unread(byte *buffer*, int *offset*, int *numChars*)

The first form pushes back the low-order byte of *ch*. This will be the next byte returnedby a subsequent call to read( ). The second form returns the bytes in *buffer*. The thirdform pushes back *numChars* bytes beginning at *offset* from *buffer*. An IOException will be thrown if there is an attempt to return a byte when the pushback buffer is full.Java 2 made a small change to PushbackInputStream: it added the skip( ) method.Here is an example that shows how a programming language parser might use aPushbackInputStream and unread( ) to deal with the difference between the = =operator for comparison and the = operator for assignment:

```
// Demonstrate unread().
import java.io.*;
class PushbackInputStreamDemo {
public static void main(String args[]) throws IOException {
String s = "if (a == 4) a = 0;\n";
byte buf[] = s.getBytes();
ByteArrayInputStream in = new ByteArrayInputStream(buf);
PushbackInputStream f = new PushbackInputStream(in);
int c;
while ((c = f.read()) != -1) {
switch(c) {
```

```
case '=':

if ((c = f.read()) == '=')
System.out.print(".eq.");
else {
System.out.print("<-");
f.unread(c);
}
break;
default:
System.out.print((char) c);
break;
}
}
}
}
```

Here is the output for this example. Notice that = = was replaced by “.eq.” and = was

replaced by “<–”.

```
if (a .eq. 4) a <- 0;
```

PushbackInputStream has the side effect of invalidating the mark( ) or reset( )methods of the InputStream used to create it. Use markSupported( ) to check anystream on which you are going to use mark( )/reset( ).

SequenceInputStream:

The SequenceInputStream class allows you to concatenate multiple InputStreams.The construction of a SequenceInputStream is different from any other InputStream.A SequenceInputStream constructor uses either a pair of InputStreams or anEnumeration of InputStreams as its argument:

SequenceInputStream(InputStream *first*, InputStream *second*)
SequenceInputStream(Enumeration *streamEnum*)

Operationally, the class fulfills read requests from the first InputStream until it runsout and then switches over to the second one. In the case of an Enumeration, it willcontinue through all of the InputStreams until the end of the last one is reached.Here is a simple example that uses aSequenceInputStream to output the contentsof two files:

```
// Demonstrate sequenced input.
import java.io.*;
import java.util.*;
```

```
class InputStreamEnumerator implements Enumeration {
private Enumeration files;
public InputStreamEnumerator(Vector files) {
this.files = files.elements();
}
public boolean hasMoreElements() {
return files.hasMoreElements();
}
public Object nextElement() {
try {
return new FileInputStream(files.nextElement().toString());
} catch (Exception e) {
return null;
}
}
}
class SequenceInputStreamDemo {
public static void main(String args[]) throws Exception {
int c;
Vector files = new Vector();
files.addElement("/autoexec.bat");
files.addElement("/config.sys");
InputStreamEnumerator e = new InputStreamEnumerator(files);
InputStream input = new SequenceInputStream(e);
while ((c = input.read()) != -1) {
System.out.print((char) c);
}
input.close();
}
}
```

This example creates a Vector and then adds two filenames to it. It passes that vector ofnames to the InputStreamEnumerator class, which is designed to provide a wrapperon the vector where the elements returned are not the filenames but rather openFileInputStreams on those names. The SequenceInputStream opens each file in turn,and this example prints the contents of the two files.

PrintStream:

The PrintStream class provides all of the formatting capabilities we have been usingfrom the System file handle, System.out, since the beginning of the book. Here are twoof its constructors:

PrintStream(OutputStream *outputStream*)PrintStream(OutputStream *outputStream*, boolean *flushOnNewline*)where *flushOnNewline* controls whether Java flushes the output stream every time anewline (\n) character is output. If *flushOnNewline* is true, flushing automatically takesplace. If it is false, flushing is not automatic. The first constructor does notautomatically flush.Java’s PrintStream objects support the print( ) and println( ) methods for all types,including Object. If an argument is not a simple type, the PrintStream methods willcall the object’s toString( ) method and then print the result.

RandomAccessFile:

RandomAccessFile encapsulates a random-access file. It is not derived from InputStream or OutputStream. Instead, it implements the interfaces DataInput and DataOutput, which define the basic I/O methods. It also supports positioning requests—that is, you can position the *file pointer* within the file. It has these two constructors:

RandomAccessFile(File *fileObj*, String *access*) throws FileNotFoundException

RandomAccessFile(String *filename*, String *access*) throws FileNotFoundException

In the first form, *fileObj* specifies the name of the file to open as a File object. In thesecond form, the name of the file is passed in *filename.* In both cases, *access* determineswhat type of file access is permitted. If it is “r”, then the file can be read, but notwritten. If it is “rw”, then the file is opened in read-write mode. If it is “rws”, the fileis opened for read-write operations and every change to the file’s data or metadatawill be immediately written to the physical device. If it is “rwd”, the file is opened forread-write operations and every change to the file’s data will be immediately writtento the physical device.The method seek( ), shown here, is used to set the current position of the filepointer within the file:

void seek(long *newPos*) throws IOException

Here, *newPos* specifies the new position, in bytes, of the file pointer from the beginningof the file. After a call to seek( ), the next read or write operation will occur at the newfile position. RandomAccessFile implements the standard input and output methods, whichyou can use to read and write to random access files. It also includes some additionalmethods. One issetLength ( ). It has this signature:void setLength(long *len*) throws IOExceptionThis method sets the length of the invoking file to that specified by *len*. This method can beused to lengthen or shorten a file. If the file is lengthened, the added portion is undefined.Java 2, version 1.4 added the getChannel( ) method to RandomAccessFile. This method returns a channel connected to the RandomAccessFile object. Channels are used by the new I/O classes contained in java.nio.

The Character Streams:

While the byte stream classes provide sufficient functionality to handle any type of I/Ooperation, they cannot work directly with Unicode characters. Since one of the mainpurposes of Java is to support the “write once, run anywhere” philosophy, it wasnecessary to include direct I/O support for characters. In this section, several of thecharacter I/O classes are discussed. As explained earlier, at the top of the characterstream hierarchies are the Reader and Writer abstract classes. We will begin with them.

Reader:

Reader is an abstract class that defines Java’s model of streaming character input. All ofthe methods in this class will throw an IOException on error conditions. Table 3provides a synopsis of the methods in Reader.

Writer:

Writer is an abstract class that defines streaming character output. All of the methodsin this class return a void value and throw an IOException in the case of errors.

FileReader

The FileReader class creates a Reader that you can use to read the contents of a file. Its two most commonly used constructors are shown here:

FileReader(String *filePath*)
FileReader(File *fileObj*)

Either can throw a FileNotFoundException. Here, *filePath* is the full path name of a file, and *fileObj* is a File object that describes the file.

Table 1.13

Method	Description
abstract void close( )	Closes the input source. Further readattempts will generate an IOException.
void mark(int *numChars*)	Places a mark at the current point in theinput stream that will remain valid until*numChars* characters are

	read.boolean markSupported ( ) Returns true if mark( )/reset( ) aresupported on this stream.
int read( )	Returns an integer representation of thenext availablecharacter from the invokinginput stream. –1 is returned when the endof the file is encountered.
int read(char *buffer*[ ])	Attempts to read up to *buffer*.*length*characters into *buffer* and returns the actualnumber of characters that were successfullyread. –1 is returned when the end of the fileis encountered.
abstract int read(char *buffer*[ ], int *offset,*int *numChars*)	Attempts to read up to *numChars* charactersinto *buffer* starting at *buffer*[*offset*], returningthe number of characters successfully read.–1 is returned when the end of the file isencountered.
boolean ready( )	Returns true if the next input request willnot wait.Otherwise, it returns false.
void reset( )	Resets the input pointer to the previouslyset mark.
long skip(long *numChars*)	Skips over *numChars* characters of input,returning the number of characters actuallyskipped.

The following example shows how to read lines from a file and print theseto the standard output stream. It reads its own source file, which must be in thecurrent directory.

```
// Demonstrate FileReader.
import java.io.*;
class FileReaderDemo {
public static void main(String args[]) throws Exception {
FileReader fr = new FileReader("FileReaderDemo.java");
BufferedReader br = new BufferedReader(fr);
String s;
```

Method	**Description**

abstract void close( )	Closes the output stream. Further writeattempts will generate an IOException.
abstract void flush( )	Finalizes the output state so that anybuffers are cleared. That is, it flushes theoutput buffers.
void write(int *ch*)	Writes a single character to the invokingoutput stream. Note that the parameter isan int, which allows you to call write withexpressions without having to cast themback to char.
void write(char *buffer*[ ])	Writes a complete array of characters totheinvoking output stream.
abstract void write(char *buffer*[ ], int *offset*,int *numChars*)	Writes a subrange of *numChars* charactersfrom the array *buffer*, beginning at*buffer*[*offset*] to the invoking output stream.
void write(String *str*)	Writes *str* to the invoking output streamvoid write(String *str*, int *offset*,int *numChars*) Writes a subrange of *numChars* charactersfrom the array *str,* beginning at thespecified *offset*.

```
while((s = br.readLine()) != null) {
System.out.println(s);
}
fr.close();
}
}
```

FileWriter

FileWriter creates a Writer that you can use to write to a file. Its most commonly usedconstructors are shown here:

FileWriter(String *filePath*)
FileWriter(String *filePath,* boolean *append)*
FileWriter(File *fileObj*)
FileWriter(File *fileObj*, boolean *append*)

They can throw an IOException. Here, *filePath* is the full path name of a file, and *fileObj* is a File object that describes the file. If *append* is true, then output is appended to theend of the file. The fourth constructor was added by Java 2, version 1.4.Creation of a FileWriter is not dependent on the file already existing. FileWriterwill create the file before opening it for output when you create the object. In the casewhere you attempt to open a read-only file, an IOException will be thrown.The following example is a character stream version of an example shown earlierwhen FileOutputStream was discussed. This version creates a sample buffer ofcharacters by first making a String and then using the getChars( ) method to extractthe character array equivalent. It then creates three files. The first, file1.txt, will containevery other character from the sample. The second, file2.txt, will contain the entire setof characters. Finally, the third, file3.txt, will contain only the last quarter.

```
// Demonstrate FileWriter.
import java.io.*;
class FileWriterDemo {
public static void main(String args[]) throws Exception {
String source = "Now is the time for all good men\n"
+ " to come to the aid of their country\n"
+ " and pay their due taxes.";
char buffer[] = new char[source.length()];
source.getChars(0, source.length(), buffer, 0);

FileWriter f0 = new FileWriter("file1.txt");
for (int i=0; i < buffer.length; i += 2) {
f0.write(buffer[i]);
}
f0.close();
FileWriter f1 = new FileWriter("file2.txt");
f1.write(buffer);
f1.close();
FileWriter f2 = new FileWriter("file3.txt");
f2.write(buffer,buffer.length-buffer.length/4,buffer.length/4);
f2.close();
}
}
```

CharArrayReader:

CharArrayReader is an implementation of an input stream that uses a character array asthe source. This class has two constructors, each of which requires a character arrayto provide the data source:

CharArrayReader(char *array*[ ])
CharArrayReader(char *array*[ ], int *start*, int *numChars*)

Here, *array* is the input source. The second constructor creates a Reader from a subsetof your character array that begins with the character at the index specified by *start* andis *numChars* long.The following example uses a pair of CharArrayReaders:

```
// 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();
}
}
```

The input1 object is constructed using the entire lowercase alphabet, while input2 contains only the first five letters. Here is the output:

```
input1 is:
abcdefghijklmnopqrstuvwxyz
input2 is:
abcde
```

CharArrayWriter

CharArrayWriter is an implementation of an output stream that uses an array as thedestination. CharArrayWriter has two constructors, shown here:

CharArrayWriter( )
CharArrayWriter(int *numChars*)

In the first form, a buffer with a default size is created. In the second, a buffer is createdwith a size equal to that specified by *numChars*. The buffer is held in the buf field ofCharArrayWriter. The buffer size will be increased automatically, if needed. Thenumber of characters held by the buffer is contained in the count field ofCharArrayWriter. Both buf and count are protected fields.The following example demonstrates CharArrayWriter by reworking the sampleprogram shown earlier for ByteArrayOutputStream. It produces the same output asthe previous version.

```
// 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());
}
}
```

BufferedReader:

BufferedReader improves performance by buffering input. It has two constructors:

BufferedReader(Reader *inputStream*)
BufferedReader(Reader *inputStream*, int *bufSize*)

The first form creates a buffered character stream using a default buffer size. In thesecond, the size of the buffer is passed in *bufSize*.As is the case with the byte-oriented stream, buffering an input character streamalso provides the foundation required to support moving backward in the streamwithin the available buffer. To support this, BufferedReader implements the mark(

)and reset( ) methods, and BufferedReader.markSupported( ) returns true.The following example reworks the BufferedInputStream example, shown earlier,so that it uses a BufferedReader character stream rather than a buffered byte stream.As before, it uses mark( ) and reset( ) methods to parse a stream for the HTML entityreference for the copyright symbol. Such a reference begins with an ampersand (&) andends with a semicolon (;) without any intervening whitespace. The sample input hastwo ampersands, to show the case where the reset( ) happens and where it does not.

Output is the same as that shown earlier.

```
// Use buffered input.
import java.io.*;
class BufferedReaderDemo {
public static void main(String args[]) throws IOException {
String s = "This is a &copy; copyright symbol " +
"but this is &copy not.\n";
char buf[] = new char[s.length()];
s.getChars(0, s.length(), buf, 0);
CharArrayReader in = new CharArrayReader(buf);
BufferedReader f = new BufferedReader(in);
int c;
boolean marked = false;
while ((c = f.read()) != -1) {
switch(c) {
case '&':
if (!marked) {

f.mark(32);
marked = true;
} else {
marked = false;
}
break;
case ';':
if (marked) {
marked = false;
System.out.print("(c)");
} else
System.out.print((char) c);
break;
case ' ':
if (marked) {
marked = false;
f.reset();
System.out.print("&");
} else
System.out.print((char) c);
break;
default:
if (!marked)
System.out.print((char) c);
break;
}
}
}
}
```

BufferedWriter :

A BufferedWriter is a Writer that adds a flush( ) method that can be used to ensurethat data buffers are physically written to the actual output stream. Using aBufferedWriter can increase performance by reducing the number of times data isactually physically written to the output stream.A BufferedWriter has these two constructors:

BufferedWriter(Writer *outputStream*)
BufferedWriter(Writer *outputStream*, int *bufSize*)

The first form creates a buffered stream using a buffer with a default size. In thesecond, the size of the buffer is passed in *bufSize*.

PushbackReader :

The PushbackReader class allows one or more characters to be returned to the inputstream. This allows you to look ahead in the input stream. Here are its two constructors:
PushbackReader(Reader *inputStream*)
PushbackReader(Reader *inputStream*, int *bufSize*)

The first form creates a buffered stream that allows one character to be pushed back.In the second, the size of the pushback buffer is passed in *bufSize*.PushbackReader provides unread( ), which returns one or more characters to theinvoking input stream. It has the three forms shown here:

void unread(int *ch*)
void unread(char *buffer*[ ])
void unread(char *buffer*[ ], int *offset*, int *numChars*)

The first form pushes back the character passed in *ch*. This will be the next characterreturned by a subsequent call to read( ). The second form returns the characters in*buffer.* The third form pushes back *numChars* characters beginning at *offset* from *buffer*.An IOException will be thrown if there is an attempt to return a character when thepushback buffer is full.The following program reworks the earlier PushBackInputStream example byreplacing PushBackInputStream with a PushbackReader. As before, it shows how aprogramming language parser can use a pushback stream to deal with the differencebetween the == operator for comparison and the = operator for assignment.

```
// Demonstrate unread().
import java.io.*;
class PushbackReaderDemo {
public static void main(String args[]) throws IOException {
String s = "if (a == 4) a = 0;\n";
char buf[] = new char[s.length()];
s.getChars(0, s.length(), buf, 0);
CharArrayReader in = new CharArrayReader(buf);
PushbackReader f = new PushbackReader(in);
int c;

while ((c = f.read()) != -1) {
switch(c) {
case '=':
if ((c = f.read()) == '=')
System.out.print(".eq.");
else {
System.out.print("<-");
f.unread(c);
}
break;
default:
System.out.print((char) c);
break;
}
}
}
}
```

PrintWriter :

PrintWriter is essentially a character-oriented version of PrintStream. It provides theformatted output methods print( ) and println( ). PrintWriter has four constructors:

PrintWriter(OutputStream *outputStream*)

PrintWriter(OutputStream *outputStream*, boolean *flushOnNewline*)

PrintWriter(Writer *outputStream*)
PrintWriter(Writer *outputStream*, boolean *flushOnNewline*)

where *flushOnNewline* controls whether Java flushes the output stream every timeprintln( ) is called. If *flushOnNewline* is true, flushing automatically takes place. If false,flushing is not automatic. The first and third constructors do not automatically flush.Java's PrintWriter objects support the print( ) and println( ) methods for all types,including Object. If an argument is not a simple type, the PrintWriter methods will callthe object's toString( ) method and then print out the result.

Using Stream I/O :

The following example demonstrates several of Java's I/O character stream classes andmethods. This program implements the standard wc (word count) command. Theprogram has two modes: if no filenames are provided as arguments, the programoperates on the standard input stream. If one or more filenames are specified, theprogram operates on each of them.

```
// A word counting utility.
import java.io.*;
class WordCount {
public static int words = 0;
public static int lines = 0;
public static int chars = 0;
public static void wc(InputStreamReader isr)
throws IOException {
int c = 0;
boolean lastWhite = true;
String whiteSpace = " \t\n\r";
while ((c = isr.read()) != -1) {
// Count characters
chars++;
// Count lines
if (c == '\n') {
lines++;
}
// Count words by detecting the start of a word
int index = whiteSpace.indexOf(c);
if(index == -1) {
if(lastWhite == true) {
++words;
}
lastWhite = false;
}
else {
```

```
lastWhite = true;
}
}
if(chars != 0) {
++lines;
}
}
public static void main(String args[]) {

FileReader fr;
try {
if (args.length == 0) { // We're working with stdin
wc(new InputStreamReader(System.in));
}
else { // We're working with a list of files
for (int i = 0; i < args.length; i++) {
fr = new FileReader(args[i]);
wc(fr);
}
}
}
catch (IOException e) {
return;
}
System.out.println(lines + " " + words + " " + chars);
}
}
```

The wc( ) method operates on any input stream and counts the number ofcharacters, lines, and words. It tracks the parity of words and whitespace in thelastNotWhite variable.When executed with no arguments, WordCount creates an InputStreamReaderobject using System.in as the source for the stream. This stream is then passed to wc( ),which does the actual counting. When executed with one or more arguments,WordCount assumes that these are filenames and creates FileReaders for each of them,passing the resultant FileReader objects to the wc( ) method. In either case, it prints theresults before exiting.

Improving wc( ) Using a StreamTokenizer :

An even better way to look for patterns in an input stream is to use another of Java'sI/O classes: StreamTokenizer. StreamTokenizer breaks up the InputStream into *tokens* that are delimited by sets ofcharacters. It has this constructor:

StreamTokenizer(Reader *inStream*)

Here *inStream* must be some form of Reader.StreamTokenizer defines several methods. In this example, we will use only a few.To reset the default set of delimiters, we will employ the resetSyntax( ) method. Thedefault set of delimiters is finely tuned for tokenizing Java programs and is thus toospecialized for this example. We declare that our tokens, or "words," are anyconsecutive string of visible characters delimited on both sides by whitespace.We use the eolIsSignificant( ) method to ensure that newline characters will bedelivered as tokens, so we can count the number of lines as well as words. It has thisgeneral form:

void eolIsSignificant(boolean *eolFlag*)

If *eolFlag* is true, the end-of-line characters are returned as tokens. If *eolFlag* is false, the end-of-line characters are ignored.

The wordChars( ) method is used to specify the range of characters that can be usedin words. Its general form is shown here:
void wordChars(int *start*, int *end*)

Here, *start* and *end* specify the range of valid characters. In the program, characters inthe range 33 to 255 are valid word characters.The whitespace characters are specified using whitespaceChars( ). It has thisgeneral form:
void whitespaceChars(int *start*, int *end*)

Here, *start* and *end* specify the range of valid whitespace characters.

The next token is obtained from the input stream by calling nextToken( ). It returnsthe type of the token.

StreamTokenizer defines four int constants: TT_EOF, TT_EOL, TT_NUMBER,and TT_WORD. There are three instance variables. nval is a public double used tohold the values of numbers as they are recognized. sval is a public String used to hold the value of any words as they are recognized. ttype is a public int indicating the type of token that has just been read by the nextToken( ) method. If the token is a word,ttype equals TT_WORD. If the token is a number, ttype equals TT_NUMBER. If thetoken is a single character, ttype contains its value. If an end-of-line condition has been encountered, ttype equals TT_EOL. (This assumes that eolIsSignificant( ) was invoked with a true argument.) If the end of the stream has been encountered, ttype equalsTT_EOF.

The word count program revised to use a StreamTokenizer is shown here:

```
// Enhanced word count program that uses a StreamTokenizer
```

```
import java.io.*;
class WordCount {
public static int words=0;
public static int lines=0;
public static int chars=0;
public static void wc(Reader r) throws IOException {
StreamTokenizer tok = new StreamTokenizer(r);
tok.resetSyntax();
tok.wordChars(33, 255);
tok.whitespaceChars(0, ' ');
tok.eolIsSignificant(true);
while (tok.nextToken() != tok.TT_EOF) {
switch (tok.ttype) {
case StreamTokenizer.TT_EOL:
lines++;
chars++;
break;
case StreamTokenizer.TT_WORD:
words++;
default: // FALLSTHROUGH
chars += tok.sval.length();
break;
}
}
}
public static void main(String args[]) {
if (args.length == 0) { // We're working with stdin
try {
wc(new InputStreamReader(System.in));
System.out.println(lines + " " + words + " " + chars);
} catch (IOException e) {};
} else { // We're working with a list of files
int twords = 0, tchars = 0, tlines = 0;
for (int i=0; i<args.length; i++) {
try {
words = chars = lines = 0;
wc(new FileReader(args[i]));
twords += words;
tchars += chars;
tlines += lines;
System.out.println(args[i] + ": " +
lines + " " + words + " " + chars);
} catch (IOException e) {
```

```
System.out.println(args[i] + ": error.");
}
}
System.out.println("total: " +
tlines + " " + twords + " " + tchars);
}
}
}
```

Serialization :

Serialization is the process of writing the state of an object to a byte stream. This isuseful when you want to save the state of your program to a persistent storage area,such as a file. At a later time, you may restore these objects by using the process ofdeserialization.Serialization is also needed to implement Remote Method Invocation (RMI). RMIallows a Java object on one machine to invoke a method of a Java object on a different machine. An object may be supplied as an argument to that remote method. The sending machine serializes the object and transmits it. The receiving machinedeserializes it.

Assume that an object to be serialized has references to other objects, which, inturn, have references to still more objects. This set of objects and the relationshipsamong them form a directed graph. There may also be circular references within thisobject graph. That is, object X may contain a reference to object Y, and object Y maycontain a reference back to object X. Objects may also contain references to themselves. The object serialization and deserialization facilities have been designed to work correctly in these scenarios. If you attempt to serialize an object at the top of an object graph, all of the other referenced objects are recursively located and serialized.

Similarly, during the process of deserialization, all of these objects and their referencesare correctly restored.An overview of the interfaces and classes that support serialization follows.

Serializable :

Only an object that implements the Serializable interface can be saved and restored by theserialization facilities. The Serializable interface defines no members. It is simply used to indicate that a class may be serialized. If a class is serializable, all of itssubclasses are also serializable.Variables that are declared as transient are not saved by the serialization facilities.Also, static variables are not saved.

Externalizable :

The Java facilities for serialization and deserialization have been designed so that much of thework to save and restore the state of an object occurs automatically. However, there are cases inwhich the programmer may need to have control over these processes. For example, it may be desirable to use compression or encryptiontechniques. The Externalizable interface is designed for these situations.The Externalizable interface defines these two methods:

void readExternal(ObjectInput *inStream*)
throws IOException, ClassNotFoundException
void writeExternal(ObjectOutput *outStream*)
throws IOException

In these methods, *inStream* is the byte stream from which the object is to be read, and*outStream* is the byte stream to which the object is to be written.

ObjectOutput :

The ObjectOutput interface extends the DataOutput interface and supports objectserialization. It defines the methods shown in Table 5. Note especially thewriteObject( ) method. This is called to serialize an object. All of these methods willthrow an IOException on error conditions.

Table 1.14

Method	Description
void close( )	Closes the invoking stream. Further writeattempts will generate an IOException.
void flush( )	Finalizes the output state so that anybuffers are cleared. That is, it flushes theoutput buffers.
void write(byte *buffer*[ ])	Writes an array of bytes to the invokingstream.
void write(byte *buffer*[ ], int *offset*, int *numBytes*)	Writes a subrange of *numBytes* bytes fromthe array *buffer*, beginning at *buffer*[*offset*].
void write(int *b*)	Writes a single byte to the invoking stream.The byte written is the low-order byte of *b*.
void writeObject(Object *obj*)	Writes object *obj* to the invoking stream.Table 5. The MethodsDefined by ObjectOutput

ObjectOutputStream:

The ObjectOutputStream class extends the OutputStream class and implements theObjectOutput interface. It is responsible for writing objects to a stream. A constructorof this class isObjectOutputStream(OutputStream *outStream*) throws IOExceptionThe argument *outStream* is the output stream to which serialized objects will be written.The most commonly used methods in this class are shown in Table. Theywill throw an IOException on error conditions. Java 2 added an inner class toObjectOuputStream called PutField. It facilitates the writing of persistent fields and itsuse is beyond the scope of this book.

Table 1.15

Method	Description
void close( )	Closes the invoking stream. Further writeattempts will generate an IOException.
void flush( )	Finalizes the output state so that anybuffers are cleared. That is, it flushes theoutput buffers.
void write(byte *buffer*[ ])	Writes an array of bytes to the invokingstream.
void write(byte *buffer*[ ], int *offset*, int *numBytes*)	Writes a subrange of *numBytes* bytes fromthe array *buffer*, beginning at *buffer*[*offset*].
void write(int *b*)	Writes a single byte to the invoking stream.The byte written is the low-order byte of *b*.
void writeBoolean(boolean *b*)	Writes a boolean to the invoking stream.
void writeByte(int *b*)	Writes a byte to the invoking stream. Thebyte written is the low-order byte of *b*.
void writeBytes(String *str*)	Writes the bytes representing *str* to theinvoking stream.
void writeChar(int *c*)	Writes a char to the invoking stream.
void writeChars(String *str*)	Writes the characters in *str* to the invokingstream.

ObjectInput :

The ObjectInput interface extends the DataInput interface and defines the methodsshown in Table. It supports object serialization. Note especially the readObject( )method. This is called to deserialize an object. All of these methods will throw anIOException on error conditions.

Table 1.16

Method	Description
void writeDouble(double *d*)	Writes a double to the invoking stream.
void writeFloat(float *f*)	Writes a float to the invoking stream.
void writeInt(int *i*)	Writes an int to the invoking stream.
void writeLong(long *l*)	Writes a long to the invoking stream.
final void writeObject (Object *obj*)	Writes *obj* to the invoking stream.
void writeShort(int i)	Writes a short to the invoking stream.
	Commonly Used Methods Defined by ObjectOutputStream (continued)

Table 1.17

Method	Description
int available( )	Returns the number of bytes that are nowavailable in the input buffer.
void close( )	Closes the invoking stream. Further readattempts will generate an IOException.
int read( )	Returns an integer representation of the nextavailable byte of input. –1 is returned whenthe end of the file isencountered.
int read(byte *buffer*[ ])	Attempts to read up to *buffer.length* bytes into*buffer,* returning the number of bytes thatwere successfully read. –1 is returned whenthe end of the file is encountered.

Methods Defined by ObjectInput

ObjectInputStream:

The ObjectInputStream class extends the InputStream class and implements theObjectInput interface.

ObjectInputStream is responsible for reading objects from a stream. A constructor of this class is objectInputStream(InputStream *inStream*) throws IOException, StreamCorruptedExceptionThe argument *inStream* is the input stream from which serialized objects should be read.The most commonly used methods in this class are shown in Table. They will throwan IOException on error conditions. Java 2 added an inner class to ObjectInputStream called GetField. It facilitates the reading of persistent fields and its use is beyond the scope of this book. Also, the method readLine( ) was deprecated by Java 2 and should no longer be used.

Table 1.18

Method	**Description**
int read(byte *buffer*[ ], int *offset*, int *numBytes*)	Attempts to read up to *numBytes* bytes into*buffer* starting at *buffer*[*offset*], returning the number of bytes that were successfully read.–1 is returned when the end of the file isencountered.
Object readObject( )	Reads an object from the invoking stream.
long skip(long *numBytes*)	Ignores (that is, skips) *numBytes* bytes in theinvoking stream, returning the number ofbytes actually ignored.

The Methods Defined by ObjectInput (continued)

Table 1.19

Method	**Description**
int available( )	Returns the number of bytes that are nowavailable in the input buffer.
void close( )	Closes the invoking stream. Further readattempts will generate an IOException.

Commonly Used Methods Defined by ObjectInputStream

Table 1.20

Method	**Description**
int read( )	Returns an integer representation of the

	nextavailable byte of input. –1 is returned whenthe end of the file is encountered.
int read(byte *buffer*[ ], int *offset*, int *numBytes*)	Attempts to read up to *numBytes* bytes into*buffer* starting at *buffer*[*offset*], returning the number of bytes successfully read. –1 is returned when the end of the file is encountered.
boolean readBoolean( )	Reads and returns a boolean from theinvoking stream.
byte readByte( )	Reads and returns a byte from theinvoking stream.
char readChar( )	Reads and returns a char from theinvoking stream.
double readDouble( )	Reads and returns a double from theinvoking stream.
float readFloat( )	Reads and returns a float from theinvoking stream.
void readFully(byte *buffer*[ ])	Reads *buffer*.*length* bytes into *buffer*. Returnsonly when all bytes have been read.
void readFully(byte *buffer*[ ], int *offset*,int *numBytes*)	Reads *numBytes* bytes into *buffer* starting at*buffer*[*offset*]. Returns only when *numBytes*have been read.
int readInt( )	Reads and returns an int from theinvoking stream.
long readLong( )	Reads and returns a long from theinvoking stream.
final Object readObject( )	Reads and returns an object from theinvoking stream.

Commonly Used Methods Defined by ObjectInputStream(continued)

A Serialization Example :

The following program illustrates how to use object serialization and deserialization.It begins by instantiating an object of class MyClass. This object has three instancevariables that are of types String, int, and double. This is the information we want tosave and restore.A FileOutputStream is created that refers to a file named “serial,” and anObjectOutputStream is created for that file stream. The writeObject( ) methodof ObjectOutputStream is then used to serialize our object. The object outputstream is flushed and closed.A FileInputStream is then created that refers to the file named “serial,” andan ObjectInputStream is created for that file stream. The readObject( ) method ofObjectInputStream is then used to deserialize our object. The object input stream isthen closed.Note that MyClass is defined to implement the Serializable interface. If this is notdone, a NotSerializableException is thrown. Try experimenting with this program bydeclaring some of the MyClass instance variables to be transient. That data is then notsaved during serialization.

```
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;
}
}
```

This program demonstrates that the instance variables of object1 and object2 are

identical. The output is shown here:

```
object1: s=Hello; i=-7; d=2.7E10
object2: s=Hello; i=-7; d=2.7E10
```

Table 1.21

Method	**Description**
short readShort( )	Reads and returns a short from the invokingstream.
int readUnsignedByte( )	Reads and returns an unsigned byte from theinvoking stream.
int readUnsignedShort( )	Reads an unsigned short from the invokingstream.

Commonly Used Methods Defined by ObjectInputStream (continued)

Stream Benefits :

The streaming interface to I/O in Java provides a clean abstraction for a complex andoften cumbersome task. The composition of the filtered stream classes allows you todynamically build the custom streaming interface to suit your data transferrequirements. Java programs written to adhere to the abstract, high-level InputStream,OutputStream, Reader, and Writer classes will function properly in the future evenwhen new and improved concrete stream classes areinvented. As you will see in thenext chapter, this model works very well when we switch from a file system-based set of streams to the network and socket streams. Finally, serialization of bjects is expected to play an increasingly important role in Java programming in the future. Java’s serialization I/O classes provide a portable solution to this sometimes tricky task.

1.13 SUMMARY

This chapter covers Object oriented programming are revisied,Java Virtual machine-,Platform independent-portability-scalability. The different types of Operators are explained and expressions-decision making,Branching, looping.

The object oriented concepts like Classes, Objects and methods are explained.

The concepts of Arrays Strings and Vectors are explained with an examples.,

Interfaces, Packages, Multi-Threading are also covered.Managing errors and exceptions,Applet programming, Managing files and streams are also covered.

1.14 QUESTIONS

1. What is JVM?
2. Explain the different data types in Java?
3. Explain left shift and Right Shift operators?
4. Explain different control statements?
5. Java is portable and scalable", Comment. 7
6. What is multithreading? Explain its need? List different Interfaces required forits implementation.
7. What is interface? Give comparison between interface and class.
8. Explain how vectors are different an array with an example.

   How will you implement multiple inheritances?
9. What is Thread? What are Thread Priorities?
10. What is package? Create your own package and use it.
11. Write a code to copy characters from one file to another.

*****

2

JAVA TECHNOLOGY FOR ACTIVE WEB DOCUMENT

ACTIVE WEB DOCUMENT

Unit Structure

2.1 INTRODUCTION

- *Active* documents consist of code executed on computer running browser

- *Java* language allows development of active document programs
  - Programs called *applets*
  - Platform independent
  - Secure

2.2 CONTINUOUS UPDATE THROUGH SERVER PUSH

- Some servers will send new versions of document
- Technique called *server push*
- Each server push document requires dedicated server resources
- May scale with number of clients
- Also requires network bandwidth for each update

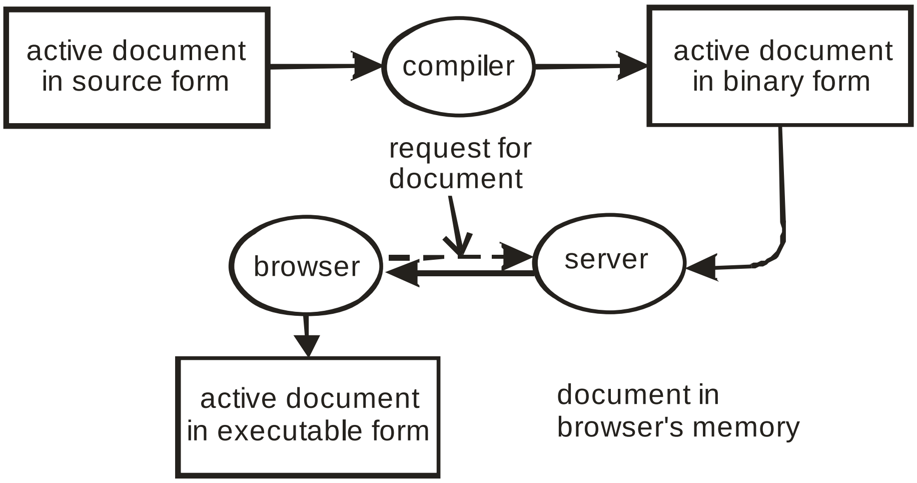
2.3 ACTIVE DOCUMENTS

- Delegates responsibility for updates to browser client
- Work scales with number of active documents on *client*
- Active document can, in fact, incur less server overhead than dynamic document

2.4 REPRESENTING AND EXECUTING ACTIVE DOCUMENTS

- Requires programming language; what should that language look like?
- How should the language be represented for execution?

Fig. 2.1


active document in source form
compiler
active document in binary form
request for document
browser
server
active document in executable form
document in browser's memory

- Multiple implementations possible; standards necessary for coordination

2.5 JAVA

- Technique for writing, compiling, downloading and executing active documents
- Includes:

  - Programming language
  - Runtime environment
  - Class library

2.6 JAVA LANGUAGE

- Resembles C++
  - Object-oriented
  - Some C++ cruft excised or restricted
- Characteristics:
  - High level, general purpose, object oriented
  - Dynamic
  - Strongly typed, static type checking
  - Concurrent

2.7 JAVA RUN-TIME ENVIRONMENT

- Interpretive execution - Java language compiled into *bytecodes*
- Automatic garbage collection
- Multithreaded execution
- Internet access
- Graphics

2.8 PORTABILITY

- Java must be platform-independent
- Run-time environment clearly defined and has no implementation dependencies
- Bytecode representation is platform independent

2.9 JAVA LIBRARY

- Library provides collection of common functions for Java applets
- Class definitions and methods
- Classes:
  - Graphics
  - Low-level network I/O (socket-level)
  - Web server interaction
  - Run-time system calls
  - File I/O
  - Data structures
  - Event capture - user interaction
  - Exception handling

2.10 AWT GRAPHICS

- Java graphics library called *Abstract Window Toolkit* (AWT)
- Includes high-level and low-level facilities
    - Windows with components - scrollbars, buttons
    - Blank rectangular area with object drawing
- Implements abstract functions; run-time environment implementation maps to window system-specific functions

2.11 JAVA AND BROWSERS

- Browser must include Java interpreter
- Java interpreter works through browser
    - Graphics
    - HTML
- Interpreter also works through native operating system
    - File I/O
    - Network operations

2.12 COMPILING A JAVA PROGRAM

- `javac` translates Java source code into bytecodes
    - Checks for correct syntax
    - Imports classes from library
    - Writes bytecode program to *filename*`.class`
- Other development environments exist
    - May include source code management
    - "Visual" systems provide "pluggable" modules

2.13 A JAVA EXAMPLE

- Example code...

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

Public class clickcount extends Applet {
    int count;
    TextField f;

Public void init(){
    count = 0;
    add(new Button(“click here”));
    f = new TextField (“the button has not been clicked at all.”);
    f.set Editable(false);
    add (f);
```

```
}

Public boolean action(Event e, Object arg) {
     if button(((Button)e.target.getlabel();==“click here’’){
        count+=1;
        f.set Text((“The button has been clicked + count +“ times’’);
         }
        return true;
        } }
```

2.14 INTERACTING WITH THE BROWSER

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

Public class buttons extends Applet {

Public void init() {
  add(new Button(“Ying”));
  add(new Button(“Yang”));
}

Public boolean action(Event e, Object arg){
   If (((Button) e.target).getLabel() == “Ying”){
      try{
        getAppletContext().showDocumet(new
        URL(http://www.nonexist.com/ying));

    }
      catch(Expection ex ) {
      //note:code to handle the exception goes here //
      }
    }
    else if (((Button) e.target).getLabel() ==“Yang”) {
      try {
           getAppletContext().showDocument(new
           URL(http://www.other.com/yang));
      }
      catch(Expection ex ) {
      //note:code to handle the exception goes here //
      }
      }
      return true;
}

}
```

2.15 ALTERNATIVES

- JavaScript
  - Interpreted scripting language
  - Like `csh` for browser

- Other languages compiled into Java bytecodes
- Other programming technologies - Inferno

2.16 JAVASCRIPT EXAMPLE

```
<SCRIPT LANGUAGE="JavaScript">
<!--
var loc = location.href.toString()
var ep = loc.lastIndexOf("/")
var dirloc = ""
if (ep > 0) {
            dirloc = loc.substring(0, ep)
            }
function doform(form)
{
// Find next file
   var URL
   URL = dirloc + "/page20b.htm"
// Some problems getting background to look nice...
   parent.frames[2].document.open()
   parent.frames[2].document.clear()

parent.frames[2].document.writeln('<HTML><HEAD></HEAD><
BODY BGCOLOR="#FFFFFF">')
   parent.frames[2].document.writeln("<DL>")
//search stuff goes here
   URL = URL + "?" + form.keyword.value
   parent.frames[0].location.href = URL
}
// some hackery to force the bottom frame to be empty every
// time the page is reloaded
   var URL
   var loc = location.href.toString()
   var ep = loc.lastIndexOf("/")
   URL = dirloc + "/page20y.htm"
// force frame 0 to empty frame so re-execution is a no-op
   parent.frames[0].location.href = URL
// -->
</SCRIPT>
```

2.17 SUMMARY

- *Active* documents execute code in browser on user's computer

- *Java* is most widely used active document technology
- Java consists of:
  - Programming language
  - Run-time environment
  - Class library
  - Tags in browser for Java program invocation

2.18 QUESTIONS

1. What is active document? How active documents are written?
2. What is the advantage of using Active document? Give example for it.
3. What is MIB? List and explain all the MIB variables with some reference.
4. List and explain all the MIB variables with some reference.

*****

3

RPC AND MIDDLEWARE

Unit Structure

3.1 INTRODUCTION

Client-server model most often used for networked applications
Fits well with program execution model
Modular
May not be easy to program
Model not intuitive to coding experience
Lots of details to manage

3.2 TOOLS FOR NETWORKED APPLICATIONS

Lots of details to manage
Connections between components of networked application
Synchronization
Robust data exchange

Data conversions

Error conditions

Many details similar or identical in different networked applications

Idea: use tools to handle routine parts of interface

3.3 PROGRAMMING WITH PROCEDURES

Modularizes code

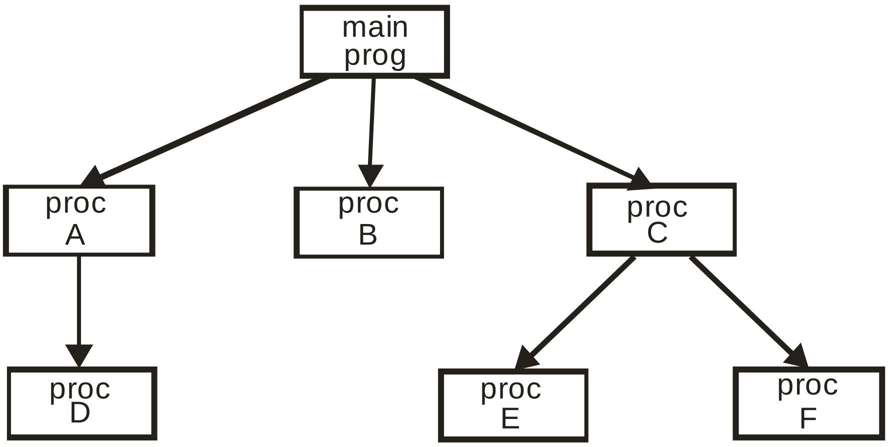
Reusable components

Procedures operate on parameters

Procedures may call other procedures

3.4 PROCEDURE CALL GRAPH

Fig. 3.1



3.5 REMOTE PROCEDURE CALL

Remote Procedure Call (RPC) model provides structure for development of networked applications

Based on procedure call model in familiar languages

Related to client-server model

Hides details of communication and synchronization beneath procedure call interface

3.6 RPC PARADIGM

Network communication hidden by procedure calls

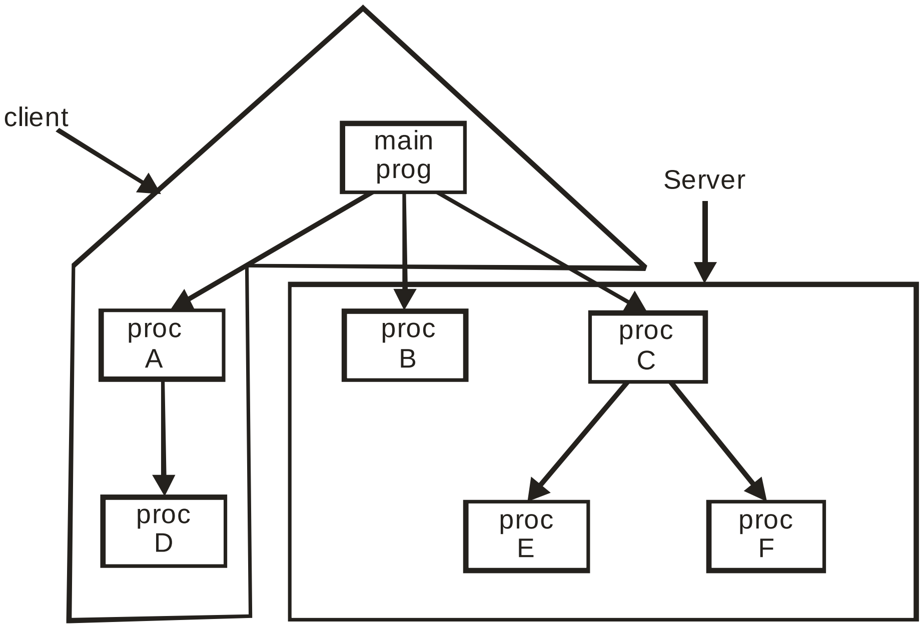
Procedure *may* be executed on different computer

RPC mechanism hides details

Programmer decides which procedures are executed where

3.7 RPC CALL GRAPH

Fig. 3.2



3.8 WHAT DOES RPC MECHANISM HAVE TO DO?

Caller:

Marshal arguments

Transmit procedure identifier and arguments to remote procedure

Wait for response

Called procedure:

Unpack arguments

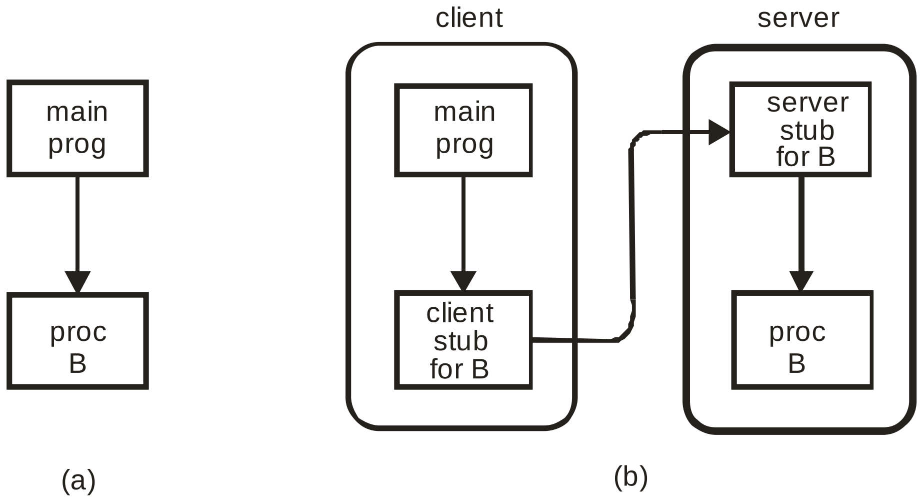
Execute procedure

Reply with any returned value

3.9 WHERE IS THIS WORK DONE?

Client stub with name of remote procedure on client
Server stub "spoofs" caller on server

Fig. 3.3



3.10 EXTERNAL DATA REPRESENTATION

Machine-dependent data representations

Integer byte ordering

Floating point

Character strings (ASCII - EBCDIC; null-terminated - length encoded)

External data representation: common format for data exchange

3.11 MIDDLEWARE AND OBJECT-ORIENTED MIDDLEWARE

Middleware: tools for generating RPC-based applications

Interface Definition Language (IDL)

Defines specifications for procedure interfaces

Used by client and server programmers

Object-oriented middleware: remote invocation of object *methods*

3.12 ONC RPC

Open Network Computing Remote Procedure Call (ONC RPC)

Designed by Sun Microsystems

Early example of middleware

Used in many Sun applications; e.g. NFS

Includes *eXternal Data Representation* (XDR) standard

3.13 DCE RPC

Open Software Foundation (OSF) defined *Distributed Computing Environment* (DCE)

Includes *DCE RPC* as middleware component

Defines its own IDL

Microsoft derived *Microsoft Remote Procedure Call* (MSRPC) from DCE RPC

3.14 CORBA

Common Object Request Broker Architecture (CORBA)

Developed by *Object Management Group* (OMG)

Vendor-independent, interoperable spec for middleware

Remote invocation instantiated by local object proxies

Proxy’s instantiated at runtime

Methods passed to "real" remote object

3.15 SUMMARY

Client-server programming may be difficult due to details and synchronization

Procedure call model is "natural" paradigm for programmers

Remote Procedure Call (RPC) uses procedure call paradigm for client-server communication

Client sends parameters for procedure call to server

Server executes procedure and replies with returned value

Middleware is category of development tools for RPC

3.16 QUESTIONS

1. Explain Remote Procedure Call Paradigm.
2. Explain RPC Paradigm.
3. Explain Communication Stubs.
4. Give external data representation.
5. Write a note on Middleware and Object-Oriented Middleware.

*****

4

NETWORK MANAGEMENT: SNMP

Unit Structure

4.1 INTRODUCTION

- Network management is a **hard** problem
- Will discuss network management paradigm base on network communication and client-server model
- SNMP is TCP/IP standard

4.2 INTERNET MANAGEMENT

- *Network manager* or *network administrator* is responsible for monitoring and controlling network hardware and software
    - Designs and implements efficient and robust network infrastructure

  - Identifies and corrects problems as they arise
  - Must know both hardware and software
- Why is network management hard?
  - Most internets *heterogeneous*
  - Most internets *large*

4.3 TYPES OF PROBLEMS

- Catastrophic
  - Fiber broken by backhoe
  - LAN switch loses power
  - Invalid route in router
  - *Easiest* to diagnose
- Intermittent or partial
  - NIC sends frames too close together
  - Router has one invalid entry
  - *Hardest* to diagnose

4.4 PROBLEM WITH HIDDEN FAILURES

- Some intermittent of partial failures may not be evident to user
  - Hardware may drop frames with data errors
  - Network protocols may recover from lost packet
- However, network performance decreases

4.5 NETWORK MANAGEMENT SOFTWARE

- Monitor operation and performance of network devices:
  - Hosts
  - Routers
  - Bridges, switches
- Control operations through rebooting, changing routing table entries

4.6 NETWORK MANAGEMENT MODEL

- Network management does *not* have an internet or transport layer protocol
- Defines application layer protocol using TCP/IP transport layer protocol
- Based on client-server model; names changes
  - *Manager* == client; run by network manager
  - *Agent* == server; runs on managed device

- Manager composes requests for agent; agent composes response and returns to manager

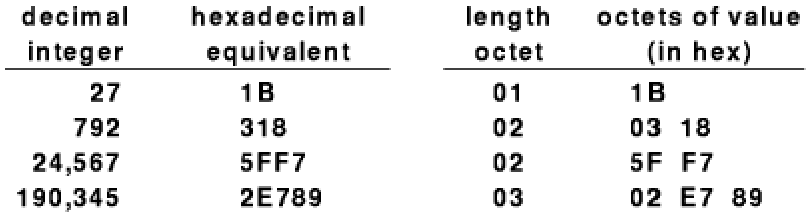
4.7 SNMP

- TCP/IP standard is *Simple Network Management Protocol* (SNMP)
- Defines all communication between manager and agent
  - Message formats
  - Interpretation of messages
  - Data representation

4.8 SNMP DATA REPRESENTATION

- SNMP uses *Abstract Syntax Notation.1* (ASN.1)
  - Platform-independent data representation standard
  - Strongly-typed
  - Can accommodate arbitrary data types
- Example - integer representation
  - Length octet - number of octets containing data
  - Data octets - value in big-endian binary

Table 4.1



decimal integer	hexadecimal equivalent	length octet	octets of value (in hex)
27	1B	01	1B
792	318	02	03 18
24,567	5FF7	02	5F F7
190,345	2E789	03	02 E7 89

4.9 FETCH-STORE PARADIGM

- Manager-agent interaction based on *fetch-store* paradigm
  - *Fetch* retrieves a value from the agent
  - *Store* changes a value on the agent
  - Any other information is extracted from the *fetch*ed data and displayed by the manager
- *Fetch* used to monitor internal data values and data structures
- *Store* used to modify and control data values and data structures; also used to control behavior by setting "reboot" object

4.10 SNMP OPERATIONS

- *Get* (fetch) retrieves value of object
- *Set* (store) stores new values into object
- *Get-next* retrieves *next* object (for scanning)

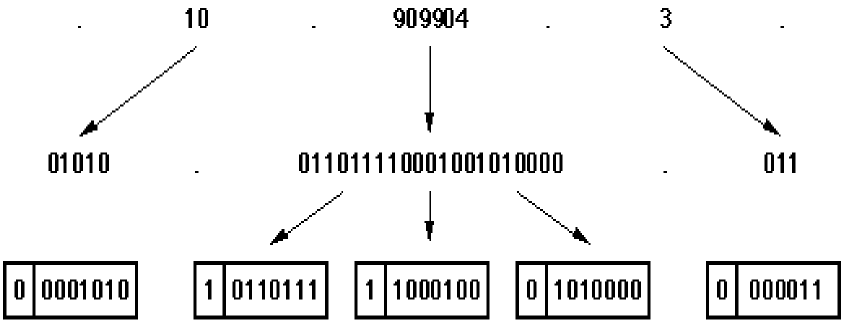
4.11 IDENTIFYING OBJECTS WITH SNMP

- SNMP is not tied to any particular set of data structures
- Operates on a collection of related objects identified in a *Management Information Base* (MIB)
- Objects in a MIB are identified by ASN.1 naming scheme
    - Hierarchical naming structure
    - Authority for new names delegated as in DNS
- Example - count of incoming IP datagrams: iso.org.dod.internet.mgmt.mib.ip.ipInReceives
- For efficiency, each name has a numeric equivalent; e.g.: 1.3.6.1.2.1.4.3

4.12 STORING ASN.1 NUMERIC VALUES

- Value stored in sequence of octets
- Leftmost bit is 0 in *last* octet
- Example:

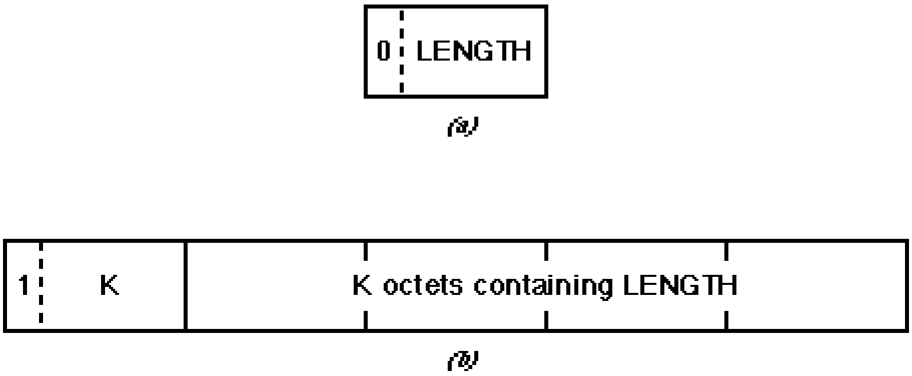
Fig. 4.1



4.13 STORING ASN.1 LENGTHS

- Leftmost bit 0 means length in same octet
- Leftmost bit 1 means length in *k* octets

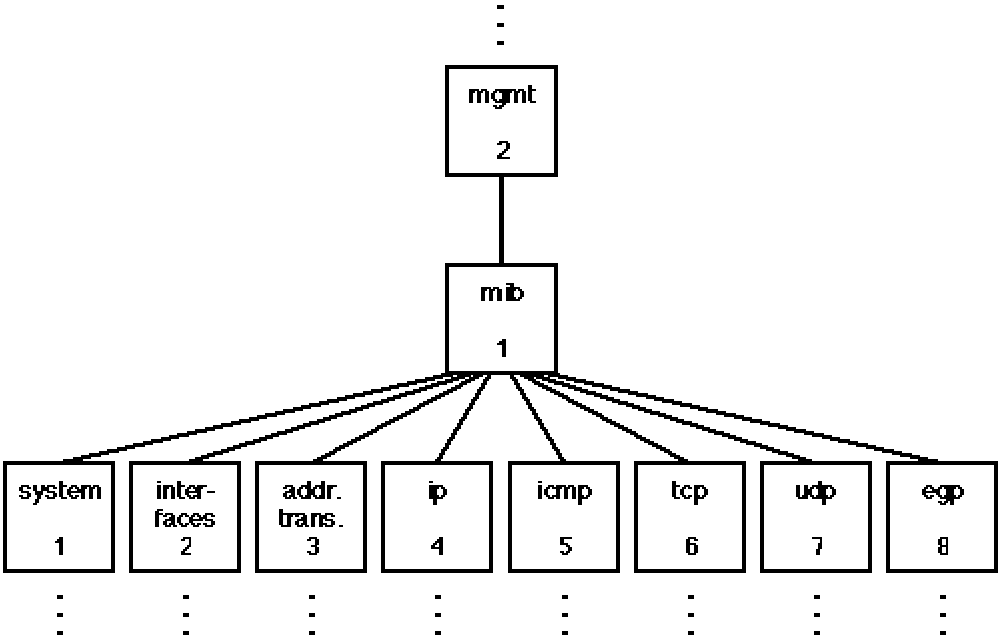
Fig. 4.2



4.14 TYPES OF MIBS

- Very flexible structure
- MIBs defined for protocols, devices, network interfaces
- *MIB I* is original TCP/IP standard for protocol suite; *MIB II* extends that original version

Fig. 4.3



4.15 ARRAYS IN MIBS

- Some types of data - such as a routing table - is most naturally stored as an array
- ASN.1 supports variable length, associative arrays
  - Number of elements can increase and decrease over time
  - Each element can be a structured object
- Indexing is implicit
  - Manager must know object is an array

    - Manager must include indexing information as suffix

4.16 ARRAY EXAMPLE

- Routing table is an array:
- ip.ipRoutingTable
- List of routing table entries is indexed by IP address
- To identify one value:
    - ip.ipRoutingTable.ipRouteEntry.ipRouteNextHop.IPdestaddr
    - ipRouteEntry indicates indexing
    - ipRouteNextHop is a field in a routing table entry
    - IPdestaddr is 32-bit IP address

4.17 SUMMARY

- TCP/IP includes *SNMP* as network management protocol
- SNMP is an *application protocol* that uses UDP for transport
- Based on *fetch-store* paradigm
    - Controls operation as side-effect of *store* operations
    - *Get-next* used top scan objects
- *Management Information Base* (MIB) defines structure of objects
- *Abstract Syntax Notation.1* (ASN.1) used for data representation and object identification

4.18 QUESTIONS

1. Explain Network management model
2. What do you mean by SNMP. Explain its data representation.
3. Explain Fetch-store paradigm.
4. What are the different Types of MIBs.
5. Explain Arrays in MIBs.

*****

5

JAVA TECHNOLOGIES GRAPHICS, IMAGES, JAVA 2D GRAPHICS

Unit Structure

5.1 JAVA GRAPHICS

The Abstract Windowing Toolkit (AWT) provides an Application Programming Interface (API)for common User Interface components, such as buttons and menus One of the main goals of Java is to provide a platform-independent development environment. The area of Graphical User Interfaces has always been one of the stickiest parts of creating highly portable code. The Windows API is different from the OS/2 Presentation Manager API, which is different from the X-Windows API, which is different from the Mac API. The most common solution to this problem is to look at all the platforms you want to use, identify the components that are common to all of them (or would be easy to implement on all of them),and create a single API you can use for all of them. On each different platform, the common API would interface with the platform’s native API. Applications using the common API would then have the same look and feel as applications using the native API. The opposite of this approach is to create a single look and feel, and then implement that look and feel on each different platform. For Java, Sun chose the common API approach, which allows Java applications to blend in smoothly with their surroundings. Sun called this common API the Abstract Windowing Toolkit, or AWT for short.

The AWT addresses graphics from two different levels. At the lower level, it handles the raw graphics functions and the different input devices such as the mouse and keyboard. At the higher level, it provides a number of components like pushbuttons and scroll bars you would otherwise have to write yourself. This chapter discusses the low-level graphics and printing features of the AWT. paint, Update, and repaint As you saw in the simple HelloWorld applet, Java applets can redraw themselves by overriding the paint method. Because your applet never explicitly calls the paint method, you may have wondered how it is called. Your applet actually has three different methods that are used in redrawing the applet, as follows:

5.2 PAINT, UPGRADE, AND REPAINT

Repaint can be called any time the applet needs to be repainted (redrawn).Update is called by repaint to signal that it is time to update the applet. The default update method clears the applet’s drawing area and calls the paint method. paint actually draws the applet’s graphics in the drawing area. The paint method is passed an instance of a Graphics class that it can use for drawing various shapes and images.

5.3 THE GRAPHICS CLASS

The Graphics class provides methods for drawing a number of graphical figures, including the following:

- Lines
- Circles and Ellipses
- Rectangles and Polygons
- Images
- Text in a variety of fonts

In addition, Graphics is extended by the Graphics2D and Graphics3D classes .

The Coordinate System:

The coordinate system used in Java is a simple Cartesian (x, y) system where x is the number of screen pixels from the left-hand side, and y is the number of pixels from the top of the screen. The upper-left corner of the screen is represented by (0, 0). This is the coordinate system used in almost all graphics systems. Figure 5.1 gives you an example of some coordinates.

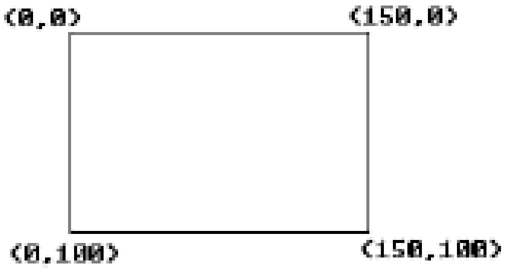


Fig 5.1

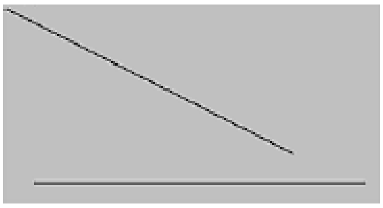
Drawing Lines:

The simplest Figure you can draw with the Graphics class is a line. The drawLine method takes two pairs of coordinates—x1,y1 and x2,y2—and draws a line between them:

```
public abstract void drawLine(int x1, int y1, int x2, int y2)
```

The applet in Listing 5.1 uses the drawLine method to draw some lines. Unlike math coordinates, where y increases from bottom to top, the y coordinates in Java increase from the top down.

Fig. 5.2 Line drawing is one of the most basic graphics operations.



The Graphics Class:

Listing 5.1 Source Code for DrawLines.java

```
import java.awt.*;
import java.applet.*;
//
// This applet draws a pair of lines using the Graphics class
//
public class DrawLines extends Applet
{
```

```
public void paint(Graphics g)
{
// Draw a line from the upper-left corner to the point at (200, 100)
g.drawLine(0, 0, 200, 100);
// Draw a horizontal line from (20, 120) to (250, 120)
g.drawLine(20, 120, 250, 120);
}
}
```

Drawing Rectangles:

To draw a rectangle, we use the drawRect method and pass it the x and y coordinates of the upper-left corner of the rectangle, the width of the rectangle, and its height:

```
public abstract void drawRect(int x, int y, int width, int height)
```

To draw a rectangle at (150, 100) that is 200 pixels wide and 120 pixels high, your call would be:

```
g.drawRect(150, 100, 200, 120);
```

The drawRect method draws only the outline of a box. If we want to draw a solid box, yo can use the fillRect method, which takes the same parameters as drawRect:

```
public abstract void fillRect(int x, int y, int width, int height)
```

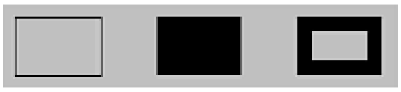
You may also clear out an area with the clearRect method, which also takes the same parameters as drawRect:

```
public abstract void clearRect(int x, int y, int width, int
height)
```

Figure 5.3 shows us the difference between drawRect, fillRect, and clearRect.

The rectangle on the left is drawn with drawRect, and the center one is drawn with fillRect. The rectangle on the right is drawn with fillRect, but the clearRect is used to make the empty area in the middle.

Fig 5.3 Java provides several flexible ways of drawing rectangles.



Drawing 3D Rectangles:

The Graphics class also provides a way to draw “3D” rectangles similar to buttons that you might find on a toolbar. Unfortunately, the Graphics class draws these buttons with very little height or depth, making the 3D effect difficult to see. The syntax for the draw3DRect andfill3DRect is similar to drawRect and fillRect, except they have an extra parameter at the end—a Boolean indicator as to whether the rectangle is raised or not:

```
public void draw3dRect(int x, int y, int width, int height, boolean raised)
public void fill3dRect(int x, int y, int width, int height, boolean raised)
```

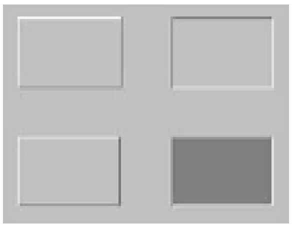
The raising/lowering effect is produced by drawing light and dark lines around the borders of the rectangle. Imagine a light coming from the upper-left corner of the screen. A raised 3D rectangle would catch light on its top and left sides, while the bottom and right sides would have a shadow. A lowered 3D rectangle would have a shadow on the top and left sides, while the bottom and right sides would catch light. Both the draw3DRect and fill3DRect methods draw the top and left sides in a lighter color for raised rectangles while drawing the bottom and right sides in a darker color. They draw the top and left darker and the bottom and right lighter for lowered rectangles. In addition, the fill3DRect method will draw the entire button in a darker shade when it is lowered. The applet in Listing 5.2 draws some raised and lowered rectangles, both filled and unfilled.

Listing 5.2 Source Code for Rect3d.java

```
import java.awt.*;’
import java.applet.*;
//
// This applet draws four varieties of 3-D rectangles.
// It sets the drawing color to the same color as the
// background because this shows up well in HotJava and
// Netscape.
public class Rect3d extends Applet
{
public void paint(Graphics g)
{
// Make the drawing color the same as the background
g.setColor(getBackground());
// Draw a raised 3-D rectangle in the upper-left
g.draw3dRect(10, 10, 60, 40, true);
// Draw a lowered 3-D rectangle in the upper-right
g.draw3dRect(100, 10, 60, 40, false);
// Fill a raised 3-D rectangle in the lower-left
g.fill3dRect(10, 80, 60, 40, true);
// Fill a lowered 3-D rectangle in the lower-right
g.fill3dRect(100, 80, 60, 40, false);
}
}
```

Figure shows the output from the Rect3d applet. Notice that the raised rectangles appear the same for the filled and unfilled. This is only because the drawing color is the same color as the background. If the drawing color were different, the filled button would be filled with the drawing color, while the unfilled button would still show the background color.

Fig. 5.4 The draw3DRect and fill3DRect methods use shading to produce a 3D effect



Drawing Rounded Rectangles :

In addition to the regular and 3D rectangles, you can also draw rectangles with rounded corners. The drawRoundRect and fillRoundRect methods are similar to drawRect and fillRectexcept that they take two extra parameters:

public abstract void drawRoundRect(int x, int y, int width, int height,int arcWidth, int arcHeight)

public abstract void fillRoundRect(int x, int y, int width, int height,int arcWidth, int arcHeight)

The arcWidth and arcHeight parameters indicate how much of the corners will be rounded.

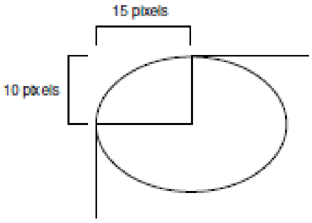
For instance, an arcWidth of 10 tells the Graphics class to round off the left-most five pixels and the right-most five pixels of the corners of the rectangle. An arcHeight of 8 tells the class to round off the top-most and bottom-most four pixels of the rectangle’s corners.

Figure 5.5 shows the corner of a rounded rectangle. The arcWidth for the Figure is 30, while the arcHeight is 10. The Figure shows an imaginary ellipse with a width of 30 and a height of 29 to help illustrate how the rounding is done.

The draw3DRect andfill3DRect methods use shading to produce a 3D effect.

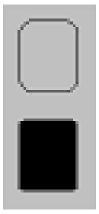
10 pixels 15 pixels

Fig. 5.5 Java uses an ellipse to determine the amount of rounding.



The applet in Listing 5.3 draws a rounded rectangle and a filled, rounded rectangle.

Fig. 5.6 Java's rounded rectangles are a pleasant alternative to sharp-cornered rectangles.



Listing 5..3 Source Code for RoundRect.java

```
import java.awt.*;
import java.applet.*;
// Example 3-RoundRect Applet
//
// This applet draws a rounded rectangle and then a
// filled, rounded rectangle.
public class RoundRect extends Applet
{
public void paint(Graphics g)
{
// Draw a rounded rectangle with an arcWidth of 20, and an
arcHeight of 20
g.drawRoundRect(10, 10, 40, 50, 20, 20);
// Fill a rounded rectangle with an arcWidth of 10, and an
arcHeight of 8
g.fillRoundRect(10, 80, 40, 50, 10, 6);
}
}
```

Drawing Circles and Ellipses:

If you are bored with square shapes, you can try your hand at circles. The Graphics class does not distinguish between a circle and an ellipse, so there is no drawCircle method. Instead, you use the drawOval and fillOval methods:

```
public abstract void drawOval(int x, int y, int width, int height)
public abstract void fillOval(int x, int y, int width, int height)
```

To draw a circle or an ellipse, first imagine that the Figure is surrounded by a rectangle that just barely touches the edges. You pass drawOval the coordinates of the upper-left corner of this rectangle. You also pass the width and height of the oval. If the width and height are the same, you are drawing a circle.

Figure 5.7 illustrates the concept of the enclosing rectangle.
FIG. 5.7 Circles and ellipses are drawn within the bounds of an imaginary enclosing rectangle.

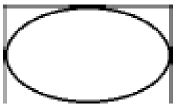
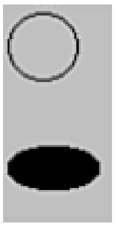


FIG. 5.8

Java doesn't know the difference between ellipses and circles; they're all just ovals.



The applet in Listing 5.4 draws a circle and a filled ellipse. Figure 5.8 shows the output from this applet. Listing 5.4 Source Code for Ovals.java

```
import java.awt.*;
import java.applet.*;
//
// This applet draws an unfilled circle and a filled ellipse
public class Ovals extends Applet
{
public void paint(Graphics g)
{
// Draw a circle with a diameter of 30 (width=30, height=30)
// With the enclosing rectangle’s upper-left corner at (0, 0)
g.drawOval(0, 0, 30, 30);
// Fill an ellipse with a width of 40 and a height of 20
// The upper-left corner of the enclosing rectangle is at (0, 60)
g.fillOval(0, 60, 40, 20);
}
}
```

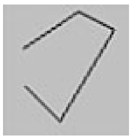
5.4 DRAWING POLYGONS

You can also draw polygons and filled polygons by using the Graphics class. You have two options when drawing polygons. You can either pass two arrays containing the x and y coordinates of the points in the polygon, or you can pass an instance of a Polygon class:

```
public abstract void drawPolygon(int[] xPoints, int[] yPoints, int numPoints)
```

```
public void drawPolygon(Polygon p)
```

FIG. 5.9 Java allows you to draw polygons of almost any shape you can imagine.



Listing 5.5 Source Code for DrawPoly.java

```
import java.applet.*;
import java.awt.*;
//
// This applet draws a polygon using an array of points
public class DrawPoly extends Applet
{
// Define an array of X coordinates for the polygon
int xCoords[] = { 10, 40, 60, 30, 10 };
// Define an array of Y coordinates for the polygon
int yCoords[] = { 20, 0, 10, 60, 40 };
public void paint(Graphics g)
{
g.drawPolygon(xCoords, yCoords, 5); // 5 points in polygon
}
}
```

The Polygon Class :

The Polygon class provides a more flexible way to define polygons. You can create a Polygon by passing it an array of x points and an array of y points:

public Polygon(int[] xPoints, int[] yPoints, int numPoints)

You can also create an empty polygon and add points to it one-at- a-time:

public Polygon()

public void addPoint(int x, int y)

Once you have created an instance of a Polygon class, you can use the getBounds method to determine the area taken up by this polygon (the minimum and maximum x and y coordinates):

public Rectangle getBounds()

The Rectangle class returned by getBounds() contains variables indicating the x and y coordinates of the rectangle and its width and height. You can also determine whether a point is contained within the polygon or outside it by calling inside with the x and y coordinates of the point:

public boolean contains(int x, int y)

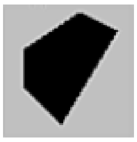
The Polygon Class

For example, you can check to see if the point (5,10) is contained within myPolygon by using the following code fragment:

```
if (myPolygon.contains(5, 10))
{// the point (5, 10) is inside this polygon
}
```

You can use this Polygon class in place of the array of points for either the drawPolygon or fillPolygon methods. The applet in Listing 5..6 creates an instance of a polygon and draws afilled polygon.

FIG. 5.10 Polygons created with the Polygon class look just like those created from an array of points.



Listing 5.6 Source Code for Polygons.java

```
import java.applet.*;
import java.awt.*;
//
// This applet creates an instance of a Polygon class and then
// uses fillPoly to draw the Polygon as a filled polygon.
public class Polygons extends Applet
{
// Define an array of X coordinates for the polygon
int xCoords[] = { 10, 40, 60, 30, 10 };
// Define an array of Y coordinates for the polygon
int yCoords[] = { 20, 0, 10, 60, 40 };
public void paint(Graphics g)
{
// Create a new instance of a polygon with 5 points
Polygon drawingPoly = new Polygon(xCoords, yCoords, 5);
// Draw a filled polygon
g.fillPolygon(drawingPoly);
}
}
```

5.5 DRAWING TEXT

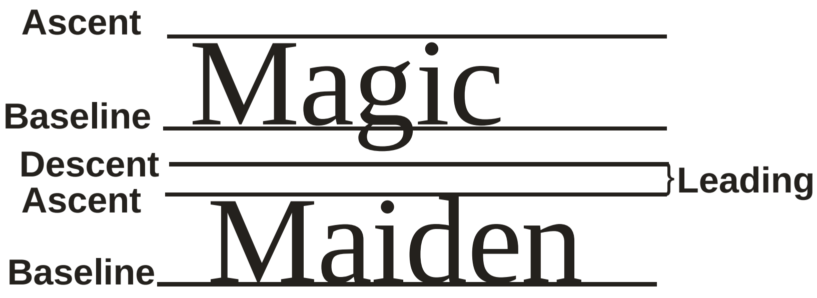
The Graphics class also contains methods to draw text characters and strings. As you have seen in the “Hello World” applet, you can use the drawString method to draw a text string on the screen. Before plunging into the various aspects of drawing text, you should be familiar with some common terms for fonts and text, as follows:

Polygons created with the Polygon class look just like those created from an array of points.
Baseline. Imaginary line the text is resting on.
Descent. How far below the baseline a particular character extends. Some characters,
such as g and j, extend below the baseline.

Ascent. How far above the baseline a particular character extends. The letter d would
have a higher ascent than the letter x.

Leading. Amount of space between the descent of one line and the ascent of the next
line. If there was no leading, such letters as g and j would almost touch such letters as M and H on the next line.
Figure 5.11 illustrates the relationship between the descent, ascent, baseline, and leading.

FIG. 5.11 Java’s font terminology originated in the publishing field, but some of the meanings have been changed.



the relationship between the descent, ascent, baseline, and leading. You may also hear the terms proportional and fixed associated with fonts. In a fixed font, every character takes up the same amount of space. Typewriters (if you actually remember those) wrote in a fixed font. Characters in a proportional font only take up as much space as they need. You can use this book as an example.

The text of the book is in a proportional font, which is much easier on the eyes. Look at some of the words and notice how the letters only take up as much space as necessary. (Compare the letters i and m, for example.) The code examples in this book, however, are written in a fixed font (this preserves the original spacing). Notice how each letter takes up exactly the same amount of space.

To draw a string using the Graphics class, you call drawString, give it the string you want to draw, and give it the x and y coordinates for the beginning of the baseline (that’s why you needed the terminology brief ing):

```
public abstract void drawString(String str, int x, int y)
```

You may recall the “Hello World” applet used this same method to draw its famous message:

```
public void paint(Graphics g)
{
g.drawString(“Hello World”, 10, 30);
}
```

You can also draw characters from an array of characters or an array of bytes. The format for drawChars and drawBytes is:

```
void drawChars(char charArray[], int offset, int numChars, int x, int y)
void drawBytes(byte byteArray[], int offset, int numChars, int x, int y)
```

The offset parameter refers to the position of the first character or byte in the array to draw. This will most often be zero because you will usually want to draw from the beginning of the array. The applet in Listing 5.7 draws some characters from a character array and from a byte array.

Listing 5.7 //Source Code for DrawChars.java

```
import java.awt.*;
import java.applet.*;
//
// This applet draws a character array and a byte array
public class DrawChars extends Applet
{
char[] charsToDraw = { ‘H’, ‘i’, ‘ ‘, ‘T’, ‘h’, ‘e’, ‘r’, ‘e’, ‘!’ };
byte[] bytesToDraw = { 65, 66, 67, 68, 69, 70, 71 }; // “ABCDEFG”
public void paint(Graphics g)
{
g.drawChars(charsToDraw, 0, charsToDraw.length, 10, 20);
g.drawBytes(bytesToDraw, 0, bytesToDraw.length, 10, 50);
}
}
```

The Font Class :

You may find that the default font for your applet is not very interesting. Fortunately, you can select from a number of different fonts. These fonts have the potential to vary from system to system, which may lead to portability issues in the future but for the moment, HotJava and Netscape support the same set of fonts. In addition to selecting between multiple fonts, you may also select a number of font styles:Font.PLAIN, Font.BOLD, and Font.ITALIC. These styles can be added together, so you can use a bold italic font with Font.BOLD + Font.ITALIC. When choosing a font, you must also give the point size of the font. Point size is a printing term that relates to the size of the font. There are 100 points to an inch when printing on a printer, but this does not necessarily apply to screen fonts. Microsoft Windows defines the point size as being about the same height in all different screen resolutions. The point sizing is done this way in Java because many applets use absolute screen coordinates, especially when drawing raw graphics. Lines and squares have a fixed pixel height. If you draw text with these figures, make the text have a fixed height as well. You create an instance of a font by using the font name, the font style, and the point size:

```
public Font(String fontName, int style, int size)
```

The following declaration creates the Times Roman font that is both bold and italic and has a

point size of 12:

```
Font myFont = new Font(“TimesRoman”, Font.BOLD + Font.ITALIC, 12);
```

You can also retrieve fonts that are described in the system properties using the getFont methods:

public static Font getFont(String propertyName)

Returns an instance of Font described by the system property named propertyName. If theproperty name is not set, it will return null.

public static Font getFont(String propertyName, Font defaultValue)

Returns an instance of Font described by the system property named propertyName. If the property name is not set, it will return defaultValue.

The getFont method allows the fonts described in the system properties to have a style and a point size associated with them in addition to the font name. The format for describing a font in the system properties is font-style-point size

The style parameter can be bold, italic, bold italic, or not present. If the style parameter is not present, the format of the string is font-point size

You might describe a bold 16-point TimesRoman font in the system properties asTimesRoman-bold-16. This mechanism is used for setting specific kinds of fonts. For instance, you might write a Java VT-100 terminal emulator that used the system property defaultVT100Font to find out whatfont to use for displaying text. You could set such a property on the command line:

```
java -DdefaultVT100Font=courier-14 emulators.vt100
```

You can get information about a font using the following methods:

public String getFamily()

Drawing Text:

The family of a font is a platform-specific name for the font. It will often be the same as the font’s name.

public String getName()

public int getSize()

public int getStyle()

You can also examine the font’s style by checking for bold, italic, and plain individually:

public boolean isBold()

public boolean isItalic()

public boolean isPlain()

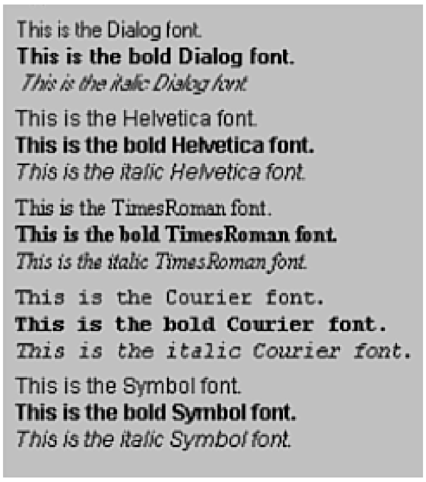
The getFontList method in the Toolkit class returns an array containing the names of the available fonts:

public abstract String[] getFontList()

You can use the getDefaultToolkit method in the Toolkit class to get a reference to the current toolkit:

public static synchronized ToolKit getDefaultToolkit()

FIG. 5.12 Java provides a number of different fonts and font styles



This is the Dialog font.
This is the bold Dialog font.
This is the italic Dialog font.
This is the Helvetica font.
This is the bold Helvetica font.
This is the italic Helvetica font.
This is the TimesRoman font.
This is the bold TimesRoman font.
This is the italic TimesRoman font.
This is the Courier font.
This is the bold Courier font.
This is the italic Courier font.
This is the Symbol font.
This is the bold Symbol font.
This is the italic Symbol font.

The applet in Listing 5.8 uses getFontList to display the available fonts in a variety of styles.

```
//Listing 5.8 Source Code for ShowFonts.java
import java.awt.*;
import java.applet.*;
//
// This applet uses the Toolkit class to get a list
// of available fonts, then displays each font in
// PLAIN, BOLD, and ITALIC style.
public class ShowFonts extends Applet
{
public void paint(Graphics g)
{
String fontList[];
Java provides a number
of different fonts and
font styles.
int i;
int startY;
// Get a list of all available fonts
fontList = getToolkit().getFontList();
startY = 15;
for (i=0; i < fontList.length; i++)
{
// Set the font to the PLAIN version
g.setFont(new Font(fontList[i], Font.PLAIN, 12));
// Draw an example
g.drawString(“This is the “+fontList[i]+” font.”, 5, startY);
// Move down a little on the screen
startY += 15;
// Set the font to the BOLD version
g.setFont(new Font(fontList[i], Font.BOLD, 12));
// Draw an example
g.drawString(“This is the bold “+fontList[i]+” font.”, 5, startY);
// Move down a little on the screen
startY += 15;
// Set the font to the ITALIC version
g.setFont(new Font(fontList[i], Font.ITALIC, 12));
// Draw an example
g.drawString(“This is the italic “+fontList[i]+” font.”, 5, startY);
// Move down a little on the screen with some extra spacing
startY += 20;
}
}
}
```

The FontMetrics Class:

The FontMetrics class lets you examine the various character measurements for a particularfont. The getFontMetrics method in the Graphics class returns an instance of FontMetrics fora particular font:

public abstract FontMetrics getFontMetrics(Font f)

You can also get the font metrics for the current font:
public FontMetrics getFontMetrics()

An instance of FontMetrics is always associated with a particular font. To find out what font an instance of FontMetrics refers to, use the getFont method:

public Font getFont()

Drawing Text

The getAscent, getDescent, getLeading, and getHeight methods return the various height aspects of a font.

public int getAscent()

returns the typical ascent for characters in the font. It is possible for certain characters in this font to extend beyond this ascent.

public int getDescent()

returns the typical descent for characters in the font. It is possible for certain characters in this font to extend below this descent.

public int getLeading()

returns the leading value for this font.

public int getHeight()

returns the total font height, calculated as ascent + descent + leading.Because some characters may extend past the normal ascent and descent, you can get theabsolute limits with

getMaxAscent and getMaxDescent:

public int getMaxAscent()

public int getMaxDescent()

The width of a character is usually given in terms of its "advance." The advance is the amount of space the character itself takes up plus the amount of white space that comes after the character.

The width of a string as printed on the screen is the sum of the advances of all its characters. The charWidth method returns the advance for a particular character:

public int charWidth(char ch)

public int charWidth(int ch)

You can also get the maximum advance for any character in the font with the getMaxAdvance method:

public int getMaxAdvance()

One of the most common uses of the FontMetrics class is to get the width, or advance, of a string of characters. The stringWidth method returns the advance of a string:

public int stringWidth(String str)

You can also get the width for an array of characters or an array of bytespublic int charsWidth(char[] data, int offset, int len)

returns the width for len characters stored in data starting at position offset.

public int bytesWidth(char[] data, int offset, int len)

returns the width for len bytes stored in data starting at position offset.

The getWidths method returns an array of widths for the first 256 characters in a font:

public int[] getWidths()

5.6 DRAWING MODES

The Graphics class has two different modes for drawing figures: paint and XOR. Paint mode means that when a Figure is drawn, all the points in that Figure overwrite the points that were underneath it. In other words, if you draw a straight line in blue, every point along that line will be blue. You probably just assumed that would happen anyway, but it doesn't have to. There is another drawing mode called XOR, short for exclusive-OR.

The XOR drawing mode dates back several decades. You can visualize how the XOR mode works by forgetting for a moment that you are dealing with colors and imagining that you are drawing in white on a black background. Drawing in XOR involves the combination of the pixelyou are trying to draw and the pixel that is on the screen where you want to draw. If you try to draw a white pixel where there is currently a black pixel, you will draw a white pixel. If you try to draw a white pixel where there is already a white pixel, you will instead draw a black pixel. This may sound strange, but it was once very common to do animation using XOR.

To understand why, you should first realize that if you draw a shape in XOR mode and then draw the shape again in XOR mode, you erase whatever you did in the first draw. If you were moving a Figure in XOR mode, you would draw it once; then to move it, you'd draw it again in its old position (thus erasing it); then XOR draws it in its new position. Whenever two objects overlapped, the overlapping areas looked like a negative: black was white and white was black.

You probably won't have to use this technique for animation, but at least you have some idea where it came from. When using XOR on a color system, think of the current drawing color as the white from the above example and identify another color as the XOR color—or the black. Because there are more than two colors, the XOR mode makes interesting combinations with other colors, but you can still erase any shape by drawing it again.

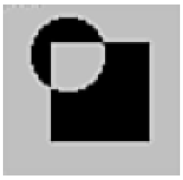
To change the drawing mode to XOR, just call the setXORMode and pass it the color you want to use as the XOR color. The applet in Listing 5..9 shows a simple animation that uses XOR mode to move a ball past a square.

Listing 5.9 Source Code for BallAnim.java

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

```
// The BallAnim applet uses XOR mode to draw a rectangle
// and a moving ball. It implements the Runnable interface
// because it is performing animation.
public class BallAnim extends Applet implements Runnable
{
Thread animThread;
int ballX = 0; // X coordinate of ball
int ballDirection = 0; // 0 if going left-to-right, 1 otherwise
// Start is called when the applet first cranks up. It creates a
thread for
// doing animation and starts up the thread.
public void start()
{
if (animThread == null)
{
animThread = new Thread(this);
animThread.start();
}
}
// Stop is called when the applet is terminated. It halts the
animation
// thread and gets rid of it.
public void stop()
{
animThread.stop();
animThread = null;
}
// The run method is the main loop of the applet. It moves the
ball, then
// sleeps for 1/10th of a second and then moves the ball again.
public void run()
{
Thread.currentThread().setPriority(Thread.NORM_PRIORITY);
while (true)
{
moveBall();
try {
Thread.sleep(100); // sleep 0.1 seconds
} catch (Exception sleepProblem) {
// This applet ignores any exceptions if it has a problem
sleeping.
// Maybe it should take Sominex
}
}
}
private void moveBall()
{
// If moving the ball left-to-right, add 1 to the x coord
if (ballDirection == 0)
{
ballX++;
// Make the ball head back the other way once the x coord hits 100
if (ballX > 100)
{
ballDirection = 1;
ballX = 100;
}
}
else
{
// If moving the ball right-to-left, subtract 1 from the x coord
ballX—;
// Make the ball head back the other way once the x coord hits 0
if (ballX <= 0)
{
ballDirection = 0;
ballX = 0;
}
}
repaint();
}
public void paint(Graphics g)
{
g.setXORMode(getBackground());
g.fillRect(40, 10, 40, 40);
g.fillOval(ballX, 0, 30, 30);
}
}
```

FIG. 5.13 XOR drawing produces an inverse effect when objects collide.



5.7 DRAWING IMAGES

The Graphics class provides a way to draw images with the drawImage method:

public abstract boolean drawImage(Image img, int x, int y, ImageObserver observer)

public abstract boolean drawImage(Image img, int x, int y, int width, int height,ImageObserver observer)

public abstract boolean drawImage(Image img, int x, int y, Color bg, ImageObserver ob)

public abstract boolean drawImage(Image img, int x, int y, int width, int height, Colorbg, ImageObserver ob)

XOR drawing produces an inverse effect when objects collide.

Drawing Images:

public abstract boolean drawImage(Image img, int dx1, int dy1, int dx2, int dy2, intsx1, int sy1, int sx2, int sy2, Color bg,ImageObserver ob)
public abstract boolean drawImage(Image img, int dx1, int dy1, int dx2,int dy2, intsx1, int sy1, int sx2, int sy2, ImageObserver ob)

The observer parameter in the drawImage method is an object that is in charge of watching to see when the image is actually ready to draw. If you are calling drawImage from within your applet, you can pass this as the observer because the Applet class implements the ImageObserver interface. The bg parameter, if present, indicates the color of the background area of the rectangle into which the image is drawn. This is often used if the image has transparent pixels where the bg color indicates the color used for the transparent pixels. The drawImage method can draw a portion of an image and scale it as it draws. The sx, sy parameters indicate the top-left and bottom-right corners of the region of the original image that is to be drawn. The dx, dy parameters indicate the top-left and bottom-right corners of theregion where the image is to be drawn. If the size of the sx and ‘ rectangles is different, the image is scaled appropriately.

To draw an image, however, you need to get the image first. That is not provided by the Graphics class. Fortunately, the Applet class provides a getImage method that you can use to retrieve images. The applet in Listing 5.10 retrieves an image and draws it

FIG. 5.14 You can draw any GIF or JPEG in a Java applet with the drawImage method.

```
import java.awt.*;
import java.applet.*;
//
// This applet uses getImage to retrieve an image
// and then draws it using drawImage
public class DrawImage extends Applet
{
private Image samImage;
public void init()
```

You can draw any GIF orJPEG in a Java appletwith the drawImagemethod.

```
{
samImage = getImage(getDocumentBase(), “samantha.gif”);
}
public void paint(Graphics g)
{
g.drawImage(samImage, 0, 0, this);
}
}
```

5.8 THE MEDIATRACKER CLASS

One problem you may face when trying to display images is that the images may be coming over a slow network link (for instance, a 14.4Kbps modem). When you begin to draw the image, it may not have arrived completely. You can use a helper class called the MediaTracker to determine whether an image is ready for display. To use the MediaTracker, you must first create one for your applet.

```
public MediaTracker(Component comp)
```

creates a new media tracker for a specific AWT component. The comp parameter is typically the applet using the media tracker.

For example, to create a media tracker within an applet:

MediaTracker myTracker = new MediaTracker(this); // “this” refers to the applet

Next, try to retrieve the image you want to display:
Image myImage = getImage(“samantha.gif”);

Now you tell the MediaTracker to keep an eye on the image. When you add an image to the MediaTracker, you also give it a numeric id:

public void addImage(Image image, int id)

The id value can be used for multiple images so when you want to see if an entire group of images is ready for display, you can check it with a single ID. If you intend to scale an image before displaying it, you should specify the intended width and height in the addImage call:

public synchronized void addImage(Image image, int id, int width, int height)

As a simple case, you can just give an image an ID of zero:myTracker.addImage(myImage, 0); // Track the image, give an id of 0

Once you have started tracking an image, you can load it and wait for it to be ready by using the waitForID method.
public void waitForID(int id)

waits for all images with an ID number of id.

The MediaTracker Class

public void waitForID(int id, long ms)

waits up to a maximum of ms milliseconds for all images with an ID number of id.

You can also wait for all images using the waitForAll method:

public void waitForAll()

As with the waitForID method, you can give a maximum number of milliseconds to wait:

public void waitForAll(long ms)

You may not want to take the time to load an image before starting your applet. You can use the statusID method to initiate a load, but not to wait for it. When you call statusID, you pass the ID you want to status and a Boolean flag to indicate whether it should start loading the image.
If you pass it true, it will start loading the image:
public int statusID(int id, boolean startLoading)

A companion to statusID is statusAll, which checks the status of all images in the

MediaTracker:

```
public int statusAll(boolean startLoading)
```

The statusID and statusAll methods return an integer that is made up of the following flags:

MediaTracker.ABORTED if any of the images have aborted loading

MediaTracker.COMPLETE if any of the images have finished loading

MediaTracker.LOADING if any images are still in the process of loading

MediaTracker.ERRORED if any images encountered an error during loading

You can also use checkID and checkAll to see if an image has been successfully loaded. All thevariations of checkAll and checkID return a Boolean value that is true if all the images checked have been loaded.

```
public boolean checkID(int id)
```

returns true if all images with a specific ID have been loaded. It does not start loading theimages if they are not loading already.

```
public synchronized boolean checkID(int id, boolean startLoading)
```

returns true if all images with a specific ID have been loaded. If startLoading is true, it will initiate the loading of any images that are not already being loaded.public boolean checkAll()

returns true if all images being tracked by this MediaTracker have been loaded, but does not initiate loading if an image is not being loaded.

```
public synchronized boolean checkAll(boolean startLoading)
```

returns true if all images being tracked by this MediaTracker have been loaded. If startLoading is true, it will initiate the loading of any images that have not started loading yet.

The applet in Listing 5.11 uses the MediaTracker to watch for an image to complete loading. It will draw text in place of the image until the image is complete; then it will draw the image.

Listing 5.11 Source Code for ImageTracker.java

```
import java.awt.*;
import java.applet.*;
import java.lang.*;
//
// The ImageTracker applet uses the media tracker to see if an
// image is ready to be displayed. In order to simulate a
// situation where the image takes a long time to display, this
// applet waits 10 seconds before starting to load the image.
// While the image is not ready, it displays the message:
// “Image goes here” where the image will be displayed.
public class ImageTracker extends Applet implements Runnable
{
Thread animThread; // Thread for doing animation
int waitCount; // Count number of seconds you have waited
MediaTracker myTracker; // Tracks the loading of an image
Image myImage; // The image you are loading
public void init()
{
// Get the image you want to show
myImage = getImage(getDocumentBase(), “samantha.gif”);
// Create a media tracker to track the image
myTracker = new MediaTracker(this);
// Tell the media tracker to track this image
myTracker.addImage(myImage, 0);
}
public void run()
{
Thread.currentThread().setPriority(Thread.NORM_PRIORITY);
while (true)
{
// Count how many times you’ve been through this loop
waitCount++;
// If you’ve been through 10 times, call checkID and tell it to start
// loading the image
if (waitCount == 10)
{
myTracker.checkID(0, true);
}
repaint();
try {
// Sleep 1 second (1000 milliseconds)
Thread.sleep(1000); // sleep 1 second
} catch (Exception sleepProblem) {
}
}
}
public void paint(Graphics g)
{
```

```
if (myTracker.checkID(0))
{
// If the image is ready to display, display it
g.drawImage(myImage, 0, 0, this);
}
else
{
// Otherwise, draw a message where you will put the image
g.drawString(“Image goes here”, 0, 30);
}
}
public void start()
{
animThread = new Thread(this);
animThread.start();
}
public void stop()
{
animThread.stop();
animThread = null;
}
}
```

5.9 GRAPHICS UTILITY CLASSES

The AWT contains several utility classes that do not perform any drawing, but represent various aspects of geometric figures. The Polygon class introduced earlier is one of these. The others are Point, Dimension, and Rectangle.

The Point Class:

A Point represents an x-y point in the Java coordinate space. Several AWT methods return instances of Point. You can also create your own instance of point by passing the x and y coordinates to the constructor:

```
public Point(int x, int y)
```

You can also create an uninitialized point, or initialize a point using another Point object:

```
public Point()
public Point(Point p)
```

The x and y coordinates of a Point object are public instance variables:

```
public int x
public int y
```

This means you may manipulate the x and y values of a Point object directly. You can also change the x and y values using either the move or translate methods:

public void move(int newX, int newY)

sets the point's x and y coordinates to newX and newY.

public void translate(int xChange, yChange)

adds xChange to the current x coordinate, and yChange to the current y.

The Dimension Class:

A dimension represents a width and height, but not at a fixed point. In other words, two rectangles can have identical dimensions without being located at the same coordinates. The empty constructor creates a dimension with a width and height of 0:

public Dimension()

You can also specify the width and height in the constructor:

public Dimension(int width, int height)

If you want to make a copy of an existing Dimension object, you can pass that object to the Dimension constructor:
public Dimension(Dimension oldDimension)

The width and height of a dimension are public instance variables, so you can manipulate them directly:

public int width

public int height

The Rectangle Class:

A rectangle represents the combination of a Point and a Dimension. The Point represents the upper-left corner of the rectangle, while the Dimension represents the rectangle's width and height. You can create an instance of a Rectangle by passing a Point and a Dimension to the constructor:
public Rectangle(Point p, Dimension d)

Rather than creating a Point and a Dimension, you can pass the x and y coordinates of the point and the width and height of the dimension:
public Rectangle(int x, int y, int width, int height)

Graphics Utility Classes

If you want x and y to be 0, you can create the rectangle using only the width and height:
public Rectangle(int width, int height)

If you pass only a Point to the constructor, the width and height are set to 0:

```
public Rectangle(Point p)
```

Similarly, if you pass only a Dimension, the x and y are set to 0:

```
public Rectangle(Dimension d)
```

You can use another Rectangle object as the source for the new rectangle’s coordinates and size:

```
public Rectangle(Rectangle r)
```

If you use the empty constructor, the x, y, width, and height are all set to 0:

```
public Rectangle()
```

The x, y, width, and height variables are all public instance variables, so you can manipulate them directly:

```
public int x
public int y
public int width
public int height
```

Like the Point class, the Rectangle class contains move and translate methods which modify the upper-left corner of the rectangle:

```
public void move(int newX, int newY)
public void translate(int xChange, yChange)
```

The setSize and grow methods change the rectangle’s dimensions in much the same way that move and translate change the upper-left corner point:

```
public void setSize(int newWidth, int newHeight)
public void grow(int widthChange, int heightChange)
```

The setBounds method changes the x, y, width, and height all in one method call:

```
public void setBounds(int newX, int newY, int newWidth, int newHeight)
```

The contains method returns true if a rectangle contains a specific x, y point:

```
public boolean contains(int x, int y)
```

The intersection method returns a rectangle representing the area contained by both the current rectangle and another rectangle:

```
public Rectangle intersection(Rectangle anotherRect)
```

You can determine if two rectangles intersect at all using the intersects method:

public boolean intersects(Rectangle anotherRect)

The union method is similar to the intersection, except that instead of returning the area in common to the two rectangles, it returns the smallest rectangle that is contained by the rectangles:

public Rectangle union(Rectangle anotherRect)

The add method returns the smallest rectangle containing both the current rectangle and another point:

public void add(Point p)

public void add(int x, int y)

If the point is contained in the current rectangle, the add method will return the current rectangle. The add method will also take a rectangle as a parameter, in which case it is identical to the union method:

public void add(Rectangle anotherRect)

5.10 THE COLOR CLASS

You may recall learning about the primary colors when you were younger. There are actually two kinds of primary colors. When you are drawing with a crayon, you are actually dealing with pigments. The primary pigments are red, yellow, and blue. You probably know some of the typical mixtures, such as red + yellow = orange, yellow + blue = green, and blue + red = purple.

Black is formed by mixing all the pigments together; while white is the absence of pigment. Dealing with the primary colors of light is slightly different. The primary colors of light are red, green, and blue. Some common combinations are red + green = brown (or yellow, depending on how bright it is), green + blue = cyan (light blue), and red + blue = magenta (purple).For colors of light, the concept of black and white are the reverse of the pigments. Black is formed by the absence of all light, while white is formed by the combination of all the primary colors. In other words, red + blue + green (in equal amounts) = white. Java uses a color model called the RGB color model. You define a color in the RGB color model by indicating how much red light, green light, and blue light is in the color. You can do this either by using numbers between zero and 255 or by using floating point numbers between 0.0 and 1.0. Table 5.1 indicates the red, green, and blue amounts for some common colors.

Table 1 Common Colors and Their RGB Values

Color Name	Red Value	Green Value	Blue Value
White	255	255	255
Light Gray	192	192	192
Gray	128	128	128
Dark Gray	64	64	64
Black	0	0	0
Red	255	0	0
Pink	255	175	175
Orange	255	200	0
Yellow	255	255	0
Green	0	255	0
Magenta	255	0	255
Cyan	0	255	255
Blue	0	0	255

You can create a custom color three ways:
Color(int red, int green, int blue)

creates a color using red, green, and blue values between zero and 255.
Color(int rgb)

creates a color using red, green, and blue values between 0 and 255, but all combined into a single integer. Bits 16–23 hold the red value, 8–15 hold the green value, and 0–7 hold the bluevalue. These values are usually written in hexadecimal notation, so you can easily see the colorvalues.

For instance, 0x123456 would give a red value of 0x12 (18 decimal), a green value of 34 (52 decimal), and a blue value of 56 (96 decimal). Notice how each color takes exactly 2 digits in hexadecimal.
Color(float red, float green, float blue)

creates a color using red, green, and blue values between 0.0 and 1.0.

Once you have created a color, you can change the drawing color using the setColor method in the Graphics class:

public abstract void setColor(Color c)

For instance, suppose you wanted to draw in pink. A nice value for pink is 255 red, 192 green, and 192 blue. The following paint method sets the color to pink and draws a circle:

public void paint(Graphics g)

{
Color pinkColor = new Color(255, 192, 192);

```
g.setColor(pinkColor);
g.drawOval(5, 5, 50, 50);
}
```

You don’t always have to create colors manually. The Color class provides a number of redefined colors:

- Color.white
- Color.lightGray
- Color.gray
- Color.darkGray
- Color.black
- Color.red
- Color.pink
- Color.orange
- Color.yellow
- Color.green
- Color.magenta
- Color.cyan
- Color.blue

Given a color, you can find out its red, green, and blue values by using the getRed, getGreen, and getBlue methods:

```
public int getRed()
public int getGreen()
public int getBlue()
```

The following code fragment creates a color and then extracts the red, green, and blue values from it:

```
int redAmount, greenAmount, blueAmount;
Color someColor = new Color(0x345678); // red=0x34, green = 0x56, blue = 0x78
redAmount = someColor.getRed(); // redAmount now equals 0x34
greenAmount = someColor.getGreen(); // greenAmount now equals 0x56
blueAmount = someColor.getBlue(); // blueAmount now equals 0x78
```

You can darken or lighten a color using the darker and brighter methods:

```
public Color darker()
public Color brighter()
```

These methods return a new Color instance that contains the darker or lighter version of the original color. The original color is left untouched.

5.11 CLIPPING

Clipping is a technique in graphics systems that prevents one area from drawing over another. Basically, you draw in a rectangular area, and everything you try to draw outside the area gets Clipping “clipped off.” Normally, your applet is clipped at the edges. In other words, you cannot draw beyond the bounds of the applet window. You cannot increase the clipping area; that is, you cannot draw outside the applet window, but you can further limit where you can draw inside the applet window. To set the boundaries of your clipping area, use the clipRect method in the

Graphics class:

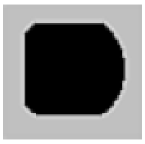
```
public abstract void clipRect(int x, int y, int width, int height)
```

You can query the current clipping area of a Graphics object with the getClipBounds method:

```
public abstract Rectangle getClipBounds()
```

The applet in Listing 5..12 reduces its drawing area to a rectangle whose upper-left corner is at (10, 10) and is 60 pixels wide and 40 pixels high, and then tries to draw a circle. Figure 5.15 shows the output from this applet.

FIG. 5.15 The clipRect method reduces the drawing area and cuts off anything that extends outside it.



Listing 5.12 Source Code for Clipper.java

```
import java.applet.*;
import java.awt.*;
//
// This applet demonstrates the clipRect method by setting
// up a clipping area and trying to draw a circle that partially
// extends outside the clipping area.
// I want you to go out there and win just one for the Clipper...
public class Clipper extends Applet
{
public void paint(Graphics g)
{
```

```
// Set up a clipping region
g.clipRect(10, 10, 60, 40);
// Draw a circle
g.fillOval(5, 5, 50, 50);
}
}
```

The clipRect method will only reduce the current clipping region. Prior to Java 1.1, there was no way to expand the clipping region once you reduced it. Java 1.1 adds the setClip method that can either expand or reduce the clipping area:
public abstract void setClip(int x, int y, int width, int height)

The clipRect method reduces the drawing area and cuts off anything that extends outside it. In preparation for the possibility of non-rectangular clipping areas, Sun has added a Shape interface and a method to use a Shape object as a clipping region. The Shape interface currently has only one method:
public abstract Rectangle getBounds()

You can set the clipping region with any object that implements the Shape interface using this variation of setClip:
public abstract void setClip(Shape region)

Since the clipping region may one day be non-rectangular, the getClipBounds method will not be sufficient for retrieving the clipping region. The getClip method returns the current clipping region as a Shape object:
public abstract Shape getClip()

Although the Shape interface might allow you to create non-rectangular clipping regions, you cannot do it yet. The only method defined in the Shape interface returns a rectangular area.The Shape interface will need to be expanded to support non-rectangular regions.

5.12 ANIMATION TECHNIQUES

You may have noticed a lot of screen flicker when you ran the Shape Manipulator applet. It was intentionally written to not eliminate any flicker so you could see just how bad flicker can be. What causes this flicker? One major cause is that the shape is redrawn on the screen right in front of you. The constant redrawing catches your eye and makes things appear to flicker. A common solution to this problem is a technique called double-buffering. The idea behind double-buffering is that you create an off screen image, and do all your drawing to that off screen image. Once you are finished drawing, you copy the off screen image to your drawing area in one quick call so the drawing area updates immediately. The other major cause of flicker is the update method. The default update method for an applet clears the drawing area, then calls your paint method. You can eliminate the flicker caused by the screen clearing by overriding update to simply call the paint method:

```
public void update(Graphics g)
{
paint(g);
}
```

CAUTION:

There is a danger with changing update this way. Your applet must be aware that the screen has not been cleared. If you are using the double-buffering technique, this should not be a problem because you are replacing the entire drawing area with your off screen image anyway. The ShapeManipulator applet can be modified easily to support double-buffering and eliminate the screen-clear. In the declarations at the top of the class, you add an Image that will be the offscreen drawing area:

```
private Image offScreenImage;
```

Next, you add a line to the init method to initialize the offscreen image:

```
offScreenImage = createImage(size().width, size().height);
```

Finally, you create an update method that does not clear the real drawing area, but makes your paint method draw to the off screen area and then copies the off screen area to the screen

Listing 5.13 An Update Method to Support Double-Buffering

```
public void update(Graphics g)
{
// This update method helps reduce flicker by supporting off-screen drawing
// and by not clearing the drawing area first. It enables you to leave
// the original paint method alone.
// Get the graphics context for the off-screen image
Graphics offScreenGraphics = offScreenImage.getGraphics();
// Now, go ahead and clear the off-screen image. It is O.K. to clear the
// off-screen image, because it is not being displayed on the screen.
// This way, your paint method can still expect a clear area, but the
// screen won’t flicker because of it.
offScreenGraphics.setColor(getBackground());
// You’ve set the drawing color to the applet’s background color,
now
// fill the entire area with that color (i.e. clear it)
offScreenGraphics.fillRect(0, 0, size().width,
size().height);
// Now, because the paint method probably doesn’t set its
drawing color,
// set the drawing color back to what was in the original graphics
context.
offScreenGraphics.setColor(g.getColor());
// Call the original paint method
paint(offScreenGraphics);
// Now, copy the off-screen image to the screen
g.drawImage(offScreenImage, 0, 0, this);
}
```

5.13 PRINTING

The ability to send information to a printer was one of the most glaring omissions in the 1.0 release of Java. Fortunately, Java 1.1 addresses that problem with the PrintJob class. The first thing you need to do in order to print something is to create an instance of a PrintJob object. You can do this with the getPrintJob method in java.awt.Toolkit:

```
public abstract PrintJob getPrintJob(Frame parent, String
jobname,Properties props)
```

As you can see, a print job must be associated with a Frame object. If you are printing from an applet, you must first create a Frame object before calling getPrintJob. Once you have a PrintJob object, you print individual pages by calling getGraphics in the PrintJob object,which creates a Graphics object that you can then draw on:

```
public abstract Graphics getGraphics()
```

Every new instance of Graphics represents a separate print page. Once you have printed all the pages you want, you call the end method in PrintJob to complete the job:

```
public abstract void end()
```

The Graphics object returned by getGraphics is identical to the Graphics object passed to your paint method. You can use all the drawing methods normally available to your paint method. In fact, you can print an image of your current screen by manually calling your paint method with the Graphics object returned by getGraphics. Once you finish drawing on a Graphics object, you invoke its dispose method to complete the page.

When printing, you often want to know the resolution of the page, or how many pixels per inchare on the page. The getResolution method in a PrintJob object returns this information:

```
public abstract int getPageResolution()
```

The getPageDimension method returns the page width and height in pixels:

```
public abstract Dimension getPageDimension()
```

Some systems and some printers print the last page first. You can find out if you will be printing in last-page-first order by calling lastPageFirst:

```
public abstract boolean lastPageFirst()
```

Listing 5.14 shows the printing equivalent of the famous “Hello World” program.

Listing 5.14 Source Code for PrintHelloWorld.java

```
import java.awt.*;
import java.applet.*;
public class PrintHelloWorld extends Applet
{
public void init()
{
// First create a frame to be associated with the print job
Frame myFrame = new Frame();
// Start a new print job
PrintJob job = Toolkit.getPrintJob(myFrame, “Hello”, NULL);
// Get a graphics object for drawing
Graphics g = job.getGraphics();
// Print the famous message to the graphics object
g.drawString(“Hello World!”, 50, 100);
// Complete the printing of this page by disposing of the graphics
object
g.dispose();
// Complete the print job
job.end();
}
}
```

5.14 DRAWING IMAGES TO THE SCREEN

Java’s methods for manipulating images are different from some of the more conventional graphics systems. To support network-based operations, Java has to support an imaging paradigm that supports the gradual loading of images.

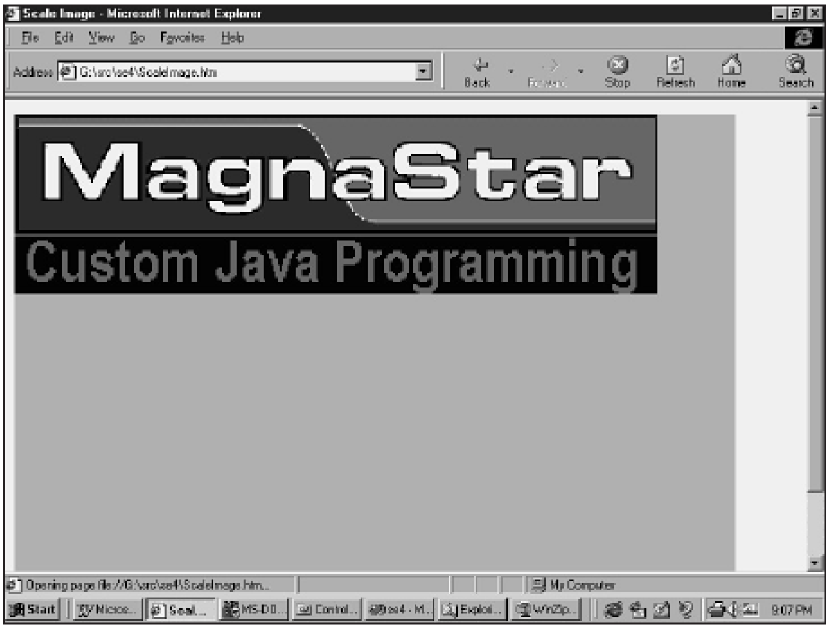
You don't want your applet to sit and wait for all the images to download. Java's producer-consumer model takes the gradual loading of images into account. Java also uses the concept of filters to enable you to change the image as it passes from producer to consumer. This might seem like a strange way to deal with images at first, but it is very powerful. Just drawing basic images to the screen is easy to do in Java. The Graphics class provides a convenient method called drawImage() for this very purpose. The drawImage method was used in several programs before. In Chapter 15, "Advanced Applet Code," "Adding Images to Applets," you learned to load images from URLs and draw them in the paint() method. The paint() method enables you to do much more than this, allowing you to scale the image dynamically.

And starting with Java 1.1, you can also crop and rotate the image just using thedrawImage method.

Listing 5.15 shows how to use the drawImage method to double the size of the image when it appears on the screen (see Figure 5.16)

.

FIG. 5.16 You can draw a simple image to the screen and scale it to twice its size.



Address G:\src\se4\ScaleImage.htm

Opening page file://G:\src\se4\ScaleImage.htm...

Listing 5.15 ScaleImage.java—Draw the Image at Twice Its Normal Size

```
import java.applet.Applet;
import java.awt.Graphics;
import java.awt.Image;
import java.net.URL;
import java.net.MalformedURLException;
public class ScaleImage extends Applet{
```

```
Image img;
public void init(){
try{
img = getImage (new
URL(getDocumentBase(),”MagnaHeader.gif”));
}catch (MalformedURLException e){
System.out.println(“URL not valid:”+e);
}
}
public void paint (Graphics g){
g.drawImage
(img,0,0,img.getWidth(null)*2,img.getHeight(null)*2,this);
}
}
```

Starting with Java 1.1, an additional drawImage method was added that provided you even more functionality. This drawImage method has the following signature:

```
public abstract boolean drawImage(Image img, int dx1,
int dy1, int dx2, int dy2,
int sx1, int sy1, int sx2, int sy2, ImageObserver observer)
```

This drawImage method works quite a bit differently from its brothers. First the initial set of parameters (the dx variables) specify not just the x and y coordinate to start from and the height and width, but the x,y coordinate of the upper-left corner of the image and the x, y coordinate of the lower-right portion of the image. This means you can actually perform axle conversions directly. The second set of parameters (the sx variables) indicate the x, y coordinates of the source to start from and end at—enabling you to crop the image at will. Look at how you can use the drawImage method in practice. Listing 5.16 shows how to use this drawImage method to flip an image upside down (see Figure 5.17).

Listing 5.16 FlipImage.java

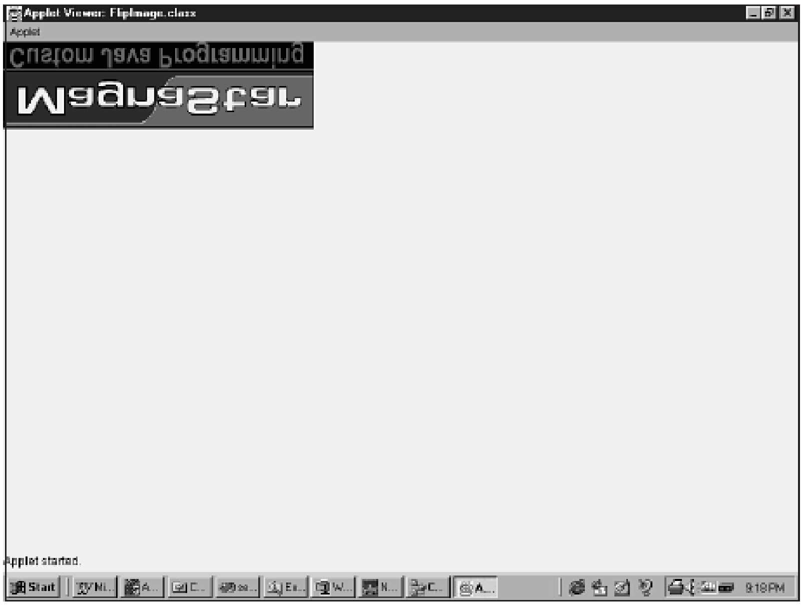
```
import java.applet.Applet;
import java.awt.Graphics;
import java.awt.Image;
import java.net.URL;
import java.net.MalformedURLException;
```

FIG. 5.1
You can draw a simple
image to the screen
and scale it to twice its
size.

```
public class FlipImage extends Applet{
Image img;
public void init(){
try{
img = getImage (new URL(getDocumentBase(),”MagnaHeader.gif”));
}catch (MalformedURLException e){
System.out.println(“URL not valid:”+e);
}
}
public void paint (Graphics g){
g.drawImage (img,0,
img.getHeight(null), img.getWidth(null), 0,0,0,
img.getWidth(null),img.getHeight(null),this);
}
}
```

Next, look at how to use the sx variables to crop out just the center of the image. In Listing 5.17 below the center of the image is drawn upside down (see Figure 5.18).

FIG. 5.17 drawImage enables you to flip an image upside–down.



Applet started.

Listing 5.17 FlipCropImage.java

```
import java.applet.Applet;
import java.awt.Graphics;
import java.awt.Image;
import java.net.URL;
import java.net.MalformedURLException;
```

```
public class FlipCropImage extends Applet{
Image img;
public void init(){
try{
img = getImage (new
URL(getDocumentBase(),”MagnaHeader.gif”));
}catch (MalformedURLException e){
System.out.println(“URL not valid:”+e);
}
}
public void paint (Graphics g){
g.drawImage (img,0, img.getHeight(null)/2,img.getWidth(null)/2,
0,img.getWidth(null)/4, img.getHeight(null)/4,
img.getWidth(null)*3/4 ,img.getHeight(null)*3/4, this);
}
}
}
```

5.15 PRODUCERS, CONSUMERS, AND OBSERVERS

Java’s model for manipulating images is more complex than other models. Java uses the concept of image producers and image consumers. An example of an image producer might be an object responsible for fetching an image over the network, or it might be a simple array ofFIG. 5.18drawImage enables you to crop out a section of the image. Producers, Consumers, and Observers bytes that represent an image. The image producer can be thought of as the source of the image data. Image consumers are objects that make use of the image data. Image consumers are, typically, low-level drawing routines that display the image onscreen. The interesting thing about the producer-consumer model is that the producer is “in control.” The ImageProducer uses the setPixels method in the ImageConsumer to describe the image to the consumer. The best way to illustrate this mechanism is to trace the process of loading an image over the network. First, the ImageProducer starts reading the image. The first thing it reads from the image is the width and height of the image. It notifies its consumers (notice that a producercan serve multiple consumers) of the dimension of the image using the setDimensionsmethod. Figure 5.19 illustrates the relationship between an ImageProducer and anImageConsumer.

Next, the producer reads the color map for the image. From this color map, the producer determines what kind of color model the image uses, and calls the setColorModel method in each consumer. Figure 25 illustrates how the producer passes color information to the consumer.

FIG. 5.19

The ImageProducer reads the image dimensions from the image file and passes the information to the ImageConsumer. The producer calls the setHints method in each consumer to tell the consumers how it intendsto deliver the image pixels. This enables the consumers to optimize their pixel handling, if possible. Some of the values for the hints are: ImageConsumer.RANDOMPIXELORDER,

ImageConsumer.TOPDOWNLEFTRIGHT, ImageConsumer.COMPLETESCANLINES, ImageConsumer. SINGLEPASS, and ImageConsumer.SINGLEFRAME. Figure 5.21 illustrates how the producer passes hints to the consumer. Now the producer finally starts to “produce” pixels, calling the setPixels method in the consumers to deliver the image. This might be done in many calls, especially if the consumers are delivering one scan line at a time for a large image. Or it might be one single call if the consumersare delivering the image as a single pass (ImageConsumer.SINGLEPASS).
Figure 5.22 shows the producer passing pixel information to the consumer.
FIG. 25

The producer uses the setColorModel method to relay color information to theconsumer. Finally, the producer calls the imageComplete method in the consumer to indicate that the image has been delivered. If a failure occurs in delivery—for instance, the network went down as it was being transmitted—then the imageComplete method will be called with a parameter of ImageConsumer.IMAGEERROR or ImageConsumer.IMAGEABORT. Another possible status is that this image is part of a multiframe image (a form of animation) and there are more frames to come.This would be signaled by the ImageConsumer.SINGLEFRAMEDONE parameter. When everything is truly complete, imageComplete is called with the ImageConsumer.STATICIMAGEDONE parameter.
Figure 5.23 shows the producer wrapping up the image transfer to the consumer.FIG. 5.21

The producer passes hints to the consumer to indicate how it will send pixels.
FIG. 5.22

The producer uses the setPixels method to pass pixel information to the consumer.

This method enables Java to load images efficiently; it does not have to stop and wait for them all to load before it begins. The ImageObserver interface is related to the producer-consumer interface as a sort of “interested third party.” It enables an object to receive updates whenever the producer has released some new information about the image. You might recall that when you used the drawImage method, you passed this as the last parameter. You were actually giving the drawImage method a reference to an ImageObserver. The Applet class implements the ImageObserver interface. The ImageObserver interface contains a single method called imageUpdate:boolean imageUpdate(Image img, int flags, int x, int y, int width, int height)

Not all the information passed to the imageUpdate method is valid all the time. The flags parameter is a summary of flags that tell what information is now available about the image. Here are the possible flags:

FIG. 5.23

The producer uses the imageComplete method to tell the consumer it is through transferring the image. Producers, Consumers, and Observers

ImageObserver.WIDTH Width value is now valid.

ImageObserver.HEIGHT Height value is now valid.

ImageObserver.PROPERTIES Image properties are now available.

ImageObserver.SOMEBITS More pixels are available (x, y, width, and height

indicate the bounding box of the pixels now available).

ImageObserver.FRAMEBITS Another complete frame is now available.

ImageObserver.ALLBITS The image has loaded completely.

ImageObserver.ERROR There was an error loading the image.

ImageObserver.ABORT The loading of the image was aborted.

These flags are usually added together, so an imageUpdate method might test for the WIDTH flag

with the following code:

```
if ((flags & ImageObserver.WIDTH) != 0) {
// width is now available
}
```

5.16 IMAGE FILTERS

The Java image model also enables you to filter images easily. The concept of a filter is similar to the idea of a filter in photography. It is something that sits between the image consumer (the film) and the image producer (the outside world). The filter changes the image before it is delivered to the consumer. The CropImageFilter is a predefined filter that crops an image to a certain dimension. (It only shows a portion of the whole image.) You create a CropImageFilter by passing the x, y, width, and height of the cropping rectangle to the constructor:

public CropImageFilter(int x, int y, int width, int height)

After you create an image filter, you can lay it on top of an existing image source by creating aFilteredImageSource:

public FilteredImageSource(ImageProducer imageSource, ImageFilter filter)

The applet in Listing 5.18 takes an image and applies a CropImageFilter to it to display only a part of the image. Figure 5.24contains the output from this applet, showing a full image and a cropped version of that image.

Listing 5.18 Source Code for CropImage.java

```
import java.awt.*;
import java.awt.image.*;
import java.applet.*;
// Example 5.4 - CropImage Applet
//
// This applet creates a CropImageFilter to create a
// cropped version of an image. It displays both the original
// and the cropped images.
public class CropImage extends Applet
{
private Image originalImage;
private Image croppedImage;
private ImageFilter cropFilter;
public void init()
{
// Get the original image
originalImage = getImage(getDocumentBase(), “samantha.gif”);
// Create a filter to crop the image in a box starting at (25, 30)
// that is 75 pixels wide and 75 pixels high.
cropFilter = new CropImageFilter(25, 30, 75, 75);
// Create a new image that is a cropped version of the original
croppedImage = createImage(new FilteredImageSource(
originalImage.getSource(), cropFilter));
}
public void paint(Graphics g)
{
// Display both images
g.drawImage(originalImage, 0, 0, this);
g.drawImage(croppedImage, 0, 200, this);
}
}
```

FIG. 5.24

The CropImageFilter enables you to display only a portion of an image.

5.17 COPYING MEMORY TO AN IMAGE

One possible type of image producer is an array of integers representing the color values of each pixel. The MemoryImageSource class is just that. You create the memory image and then create a MemoryImageSource to act as an image producer for that memory image. Next, you create an image from the MemoryImageSource. MemoryImageSource has a number of constructors.
In all of them, you must supply the width and height of the image, the array of pixel

Copying Memory to an Image

values, the starting offset of the first pixel in the array, and the number of positions that make up a scan line in the image. The pixel values are normally the RGB values for each pixel; however, if you supply your own color model, the meaning of the pixel values is determined by the color model. The scanline length is usually the same as the image width. Sometimes, however, your pixel array might have extra padding at the end of the scanline, so you might have a scanline length larger than the image width. You cannot have a scanline length shorter than the image width. You can also pass a table of properties for the image that will be passed to the image consumer. You need the properties only if you have an image consumer that requires them. The consumers that ship with the JDK do not require any properties.

Here are the constructors for the MemoryImageSource:
public MemoryImageSource(int width, int height, ColorModel model,

byte[] pixels, int startingOffset, int scanlineLength)

public MemoryImageSource(int width, int height, ColorModel model,

byte[] pixels, int startingOffset, int scanlineLength, Hashtable properties)

public MemoryImageSource(int width, int height, ColorModel model,

int[] pixels, int startingOffset, int scanlineLength)

public MemoryImageSource(int width, int height, ColorModel model,

int[] pixels, int startingOffset, int scanlineLength, Hashtable properties)

public MemoryImageSource(int width, int height, int[] pixels,

int startingOffset, int scanlineLength)

public MemoryImageSource(int width, int height, int[] pixels,

int startingOffset, int scanlineLength, Hashtable properties)

The applet in Listing 5.19. creates a memory image, a MemoryImageSource, and finally draws the image in the drawing area. Figure 5.25 shows the output from this applet.

Listing 5.19 Source Code for MemoryImage.java

```
import java.applet.*;
import java.awt.*;
import java.awt.image.*;
// Example 25 - MemoryImage Applet
//
// This applet creates an image using an array of
// pixel values.
public class MemoryImage extends Applet
{
```

FIG. 5.25 MemoryImageSource class enables you to create your own images from pixel values.

```
private final static int b = Color.blue.getRGB();
private final static int r = Color.red.getRGB();
private final static int g = Color.green.getRGB();
// Create the array of pixel values. The image will be 10x10
// And resembles a square bullseye with blue around the outside,
// green inside the blue, and red in the center.
int pixels[] = {
b, b, b, b, b, b, b, b, b, b,
b, b, b, b, b, b, b, b, b, b,
b, b, g, g, g, g, g, g, b, b,
b, b, g, g, g, g, g, g, b, b,
b, b, g, g, r, r, g, g, b, b,
b, b, g, g, r, r, g, g, b, b,
b, b, g, g, g, g, g, g, b, b,
b, b, g, g, g, g, g, g, b, b,
b, b, b, b, b, b, b, b, b, b,
b, b, b, b, b, b, b, b, b, b};
Image myImage;
public void init()
{
// Create the new image from the pixels array. The 0, 10 means start
// reading pixels from array location 0, and there is a new row of
// pixels every 10 locations.
myImage = createImage(new MemoryImageSource(10, 10, pixels, 0, 10));
}
public void paint(Graphics g)
```

```
{
// Draw the image. Notice that the width and height we give for the
// image is 10 times its original size. The drawImage method will
// scale the image automatically.
g.drawImage(myImage, 0, 0, 100, 100, this);
}
}
```

5.18 COPYING IMAGES TO MEMORY

The PixelGrabber class is sort of an inverse of the MemoryImageSource. Rather than taking an array of integers and turning it into an image, it takes an image and turns it into an array of integers. The PixelGrabber acts as an ImageConsumer. You create a PixelGrabber, give it the dimensions of the image you want and an array in which to store the image pixels, and it gets the pixels from the ImageProducer.

To grab pixels, you must first create a PixelGrabber by passing the image you want to grab, the x, y, width, and height of the area you are grabbing, an array to contain the pixel values, and the offset and scanline length for the array of pixel values:

Copying Images to Memory

public PixelGrabber(Image image, int x, int y, int width, int height,int[] pixels, int startingOffset, int scanlineLength)You can also supply an image producer instead of an image:

public PixelGrabber(ImageProducer producer, int x, int y, int width, int height,

int[] pixels, int startingOffset, int scanlineLength)

To initiate the pixel grabbing, call the grabPixels method.

public boolean grabPixels() throws InterruptedException

starts grabbing pixels and waits until it gets all the pixels. If the pixels are grabbed successfully,

it returns true. If there is an error or an abort, it returns false.

public boolean grabPixels(long ms) throws InterruptedException

starts grabbing pixels and waits a maximum of ms milliseconds for all the pixels. If the pixels are grabbed successfully, it returns true. If there is a timeout, an error, or an abort, it returns false.

You can check on the status of a pixel grab with the status method:

public synchronized int status()

The value returned by status contains the same information as the flags parameter in the imageUpdate method in ImageObserver. Basically, if the ImageObserver.ABORT bit is set in the

value, the pixel grab is aborted; otherwise, it should be okay.

The PixelGrabber is useful if you want to take an existing image and modify it. Listing 5.19 is

an applet that uses the PixelGrabber to get the pixels of an image into an array. It then enables

you to color sections of the image by picking a crayon and touching the area you want to color.

To redisplay the image, it uses the MemoryImageSource to turn the array of pixels back into an

image. The applet runs pretty slowly on a 486/100, so you need a lot of patience. It requires the

Shape class.

Listing 5.19 Source Code for Crayon.java

```
import java.applet.*;
import java.awt.*;
import java.awt.image.*;
// Example 5.6 - Crayon Applet
//
// The Crayon applet uses the PixelGrabber to create an array of pixel
// values from an image. It then allows you to paint the image using
// a set of crayons, and then redisplays the image using the
// MemoryImageSource.
// If you want to use other images with this applet, make sure that
// the lines are done in black, since it specifically looks for black
// as the boundary for an area.
// Also, beware, this applet runs very slowly on a 486/100
public class Crayon extends Applet
{
private Image coloringBook; // the original image
private Image displayImage; // the image to be displayed
private int imageWidth, imageHeight; // the dimensions of the image
// the following two arrays set up the shape of the crayons
int crayonShapeX[] = { 0, 2, 10, 15, 23, 25, 25, 0 };
int crayonShapeY[] = { 15, 15, 0, 0, 15, 15, 45, 45 };
// We use the ShapeObject class defined earlier so we can move the crayons
// to a new location easily.
private ShapeObject crayons[];
```

```
// The color class doesn’t provide a default value for brown, so
we add one.
private Color brown = new Color(130, 100, 0);
// crayonColors is an array of all the colors the crayons can be.
You can
// add new crayons just by adding to this array.
private Color crayonColors[] = {
Color.blue, Color.cyan, Color.darkGray,
Color.gray, Color.green, Color.magenta,
Color.orange, Color.pink, Color.red,
Color.white, Color.yellow, brown };
private Color currentDrawingColor; // the color we are coloring
with
private int imagePixels[]; // the memory image of the picture
boolean imageValid = false; // did we read the image in o.k.?
// blackRGB is just used as a shortcut to get to the black pixel
value
private int blackRGB = Color.black.getRGB();
public void init()
{
int i;
MediaTracker tracker = new MediaTracker(this);
// Get the image we will color
coloringBook = getImage(getDocumentBase(), “smileman.gif”);
// tell the media tracker about the image
tracker.addImage(coloringBook, 0);
// Wait for the image, if we get an error, flag the image as invalid
try {
tracker.waitForID(0);
imageValid = true;
} catch (Exception oops) {
imageValid = false;
}
// Get the image dimensions
imageWidth = coloringBook.getWidth(this);
imageHeight = coloringBook.getHeight(this);
// Copy the image to the array of pixels
resetMemoryImage();
// Create a new display image from the array of pixels
remakeDisplayImage();
// Create a set of crayons. We determine how many crayons to
create
// based on the size of the crayonColors array
crayons = new ShapeObject[crayonColors.length];
for (i=0; i < crayons.length; i++)
{
// Create a new crayon shape for each color
crayons[i] = new ShapeObject(crayonShapeX,
crayonShapeY, crayonShapeX.length);
// The crayons are lined up in a row below the image
crayons[i].moveShape(i * 30,
imageHeight + 10);
}
// Start coloring with the first crayon
currentDrawingColor = crayonColors[0];
}
// resetMemoryImage copies the coloringBook image into the
// imagePixels array.
private void resetMemoryImage()
{
imagePixels = new int[imageWidth * imageHeight];
// Set up a pixel grabber to get the pixels
PixelGrabber grabber = new PixelGrabber(
coloringBook.getSource(),
0, 0, imageWidth, imageHeight, imagePixels,
0, imageWidth);
// Ask the image grabber to go get the pixels
try {
grabber.grabPixels();
} catch (Exception e) {
// Ignore for now
return;
}
// Make sure that the image copied correctly, although we don’t
// do anything if it doesn’t.
if ((grabber.status() & ImageObserver.ABORT) != 0)
{
// uh oh, it aborted
return;
}
}
// getPixel returns the pixel value for a particular x and y
private int getPixel(int x, int y)
{
return imagePixels[y * imageWidth + x];
}
// setPixel sets the pixel value for a particular x and y
```

```
private void setPixel(int x, int y, int color)
{
imagePixels[y*imageWidth + x] = color;
}
// floodFill starts at a particular x and y coordinate and fills it, and all
// the surrounding pixels with a color. It doesn’t paint over black pixels,
// so they represent the borders of the fill.
// The easiest way to code a flood fill is by doing it recursively - you
// call flood fill on a pixel, color that pixel, then it calls flood fill
// on each surrounding pixel and so on. Unfortunately, that usually causes
// stack overflows since recursion is pretty expensive.
// This routine uses an alternate method. It makes a queue of pixels that
// it still has to fill. It takes a pixel off the head of the queue and
// colors the pixels around it, then adds those pixels to the queue. In other
// words, a pixel is really added to the queue after it has been colored.
// If a pixel has already been colored, it is not added, so eventually, it
// works the queue down until it is empty.
private void floodFill(int x, int y, int color)
{
// If the pixel we are starting with is already black, we won’t paint
if (getPixel(x, y) == blackRGB)
{
return;
}
// Create the pixel queue. Assume the worst case where every pixel in the
// image may be in the queue.
int pixelQueue[] = new int[imageWidth * imageHeight];
int pixelQueueSize = 0;
// Add the start pixel to the queue (we created a single array of ints,
// even though we are enqueuing two numbers. We put the y value in the
// upper 16 bits of the integer, and the x in the lower 16. This gives
// a limit of 65536x65536 pixels, that should be enough.)
pixelQueue[0] = (y << 16) + x;
```

```
pixelQueueSize = 1;
// Color the start pixel.
setPixel(x, y, color);
// Keep going while there are pixels in the queue.
while (pixelQueueSize > 0)
{
// Get the x and y values of the next pixel in the queue
x = pixelQueue[0] & 0xffff;
y = (pixelQueue[0] >> 16) & 0xffff;
// Remove the first pixel from the queue. Rather than move all
the
// pixels in the queue, which would take forever, just take the
one
// off the end and move it to the beginning (order doesn’t matter
here).
pixelQueueSize--;
pixelQueue[0] = pixelQueue[pixelQueueSize];
// If we aren’t on the left side of the image, see if the pixel to the
// left has been painted. If not, paint it and add it to the queue.
if (x > 0) {
if ((getPixel(x-1, y) != blackRGB) &&
(getPixel(x-1, y) != color))
{
setPixel(x-1, y, color);
pixelQueue[pixelQueueSize] =
(y << 16) + x-1;
pixelQueueSize++;
}
}
// If we aren’t on the top of the image, see if the pixel above
// this one has been painted. If not, paint it and add it to the
queue.
if (y > 0) {
if ((getPixel(x, y-1) != blackRGB) &&
(getPixel(x, y-1) != color))
{
setPixel(x, y-1, color);
pixelQueue[pixelQueueSize] =
((y-1) << 16) + x;
pixelQueueSize++;
}
}
// If we aren’t on the right side of the image, see if the pixel to
the
// right has been painted. If not, paint it and add it to the queue.
if (x < imageWidth-1) {
if ((getPixel(x+1, y) != blackRGB) &&
(getPixel(x+1, y) != color))
{
setPixel(x+1, y, color);
pixelQueue[pixelQueueSize] =
(y << 16) + x+1;
pixelQueueSize++;
}
}
// If we aren't on the bottom of the image, see if the pixel below
// this one has been painted. If not, paint it and add it to the
queue.
if (y < imageHeight-1) {
if ((getPixel(x, y+1) != blackRGB) &&
(getPixel(x, y+1) != color))
{
setPixel(x, y+1, color);
pixelQueue[pixelQueueSize] =
((y+1) << 16) + x;
pixelQueueSize++;
}
}
}
}
// remakeDisplayImage takes the array of pixels and turns it into
an
// image for us to display.
private void remakeDisplayImage()
{
displayImage = createImage(new MemoryImageSource(
imageWidth, imageHeight, imagePixels, 0, imageWidth));
}
// The paint method is written with the assumption that the
screen has
// not been cleared ahead of time, that way we can create an
update
// method that doesn't clear the screen, but doesn't need an off-
screen
// image.
public void paint(Graphics g)
{
int i;
```

```
// If we got the image successfully, draw it, otherwise, print a
message
// saying we couldn’t get it.
if (imageValid)
{
g.drawImage(displayImage, 0, 0, this);
}
else
{
g.drawString(“Unable to load coloring image.”, 0, 50);
}
// Draw the crayons
for (i=0; i < crayons.length; i++)
{
// Draw each crayon in the color it represents
g.setColor(crayonColors[i]);
g.fillPolygon(crayons[i]);
// Get the box that would enclose the crayon
Rectangle box = crayons[i].getBoundingBox();
// If the crayon is the current one, draw a black box around it, if
not,
// draw a box the color of the background around it (in case the
current
// crayon has changed, we want to make sure the old box is
erased).
if (crayonColors[i] == currentDrawingColor)
{
g.setColor(Color.black);
}
else
{
g.setColor(getBackground());
}
// Draw the box around the crayon.
g.drawRect(box.x, box.y, box.width, box.height);
}
}
// Override the update method to call paint without clearing the
screen.
public void update(Graphics g)
{
paint(g);
}
public boolean mouseDown(Event event, int x, int y)
```

```
{
int i;
// Check each crayon to see whether the mouse was clicked inside of it. If so,
// change the current color to that crayon’s color. We use the “inside”
// method to see whether the mouse x,y is within the crayon shape. Pretty
handy!
for (i=0; i < crayons.length; i++)
{
if (crayons[i].inside(x, y))
{
currentDrawingColor = crayonColors[i];
repaint();
return true;
}
}
// If the mouse wasn’t clicked on a crayon, see whether it was clicked within
// the image. This assumes that the image starts at 0, 0.
if ((x < imageWidth) && (y < imageHeight))
{
// If the image was clicked, fill that section of the image with the
// current crayon color
floodFill(x, y, currentDrawingColor.getRGB());
// Now re-create the display image because we just changed the pixels
remakeDisplayImage();
repaint();
return true;
}
return true;
}
}
```

5.19 COLOR MODELS

The image producer-consumer model also makes use of a ColorModel class. As you have seen, the images passed between producers and consumers are made up of arrays of integers. Each integer represents the color of a single pixel. The ColorModel class contains methods to extract the red, green, blue, and alpha components from a pixel value. You are probably already familiar with the red, green, and blue color components, but the alpha component might be something new to you.

The alpha component represents the transparency of a color. An alpha value of 255 means that the color is completely opaque, whereas an alpha of zero indicates that the color is completely transparent. The default color model is the RGBdefault model, which encodes the four-color components in the form 0xaarrggbb. The left-most eight bits are the alpha value; the next eight bits are the red component followed by eight bits for green and, finally, eight bits for blue. For example, a color of 0x12345678 has an alpha component of 0x12 (fairly transparent), a redcomponent of 0x34, a green component of x56, and a blue component of 0x78.

The alpha component is used only for images. You cannot use it in conjunction with the Color class. In other words, you can't use it in any of the drawing functions in theGraphics class.

Any time you need a color model and you are satisfied with using the RGBdefault model, you can use getRGBdefault:

public static ColorModel getRGBdefault()

You can extract the red, green, blue, and alpha components of a pixel using these methods:

public abstract int getRed(int pixel)

public abstract int getGreen(int pixel)

public abstract int getBlue(int pixel)

public abstract int getAlpha(int pixel)

You can find out the number of bits per pixel in a color model using getPixelSize:

public int getPixelSize()

Because many other AWT components prefer colors in RGB format, you can ask the color

model to convert a pixel value to RGB format with getRGB:

public int getRGB(int pixel)

The DirectColorModel Class

The DirectColorModel class stores the red, green, blue, and alpha components of a pixel directly in the pixel value. The standard RGB format is an example of a direct color model. The format of the pixel is determined by a set of bitmasks that tell the color model how each color is mapped into the pixel. The constructor for the DirectColorModel takes the number of bits per pixel, the red, green, and blue bit masks, and an optional alpha mask as parameters:

public DirectColorModel(int bits, int redMask, int greenMask, int blueMask)

```
public DirectColorModel(int bits, int redMask, int greenMask, int blueMask, int alphaMask)
```

You can query the mask values using the following methods:

```
public final int getRedMask()
public final int getGreenMask()
public final int getBlueMask()
public final int getAlphaMask()
```

The bits in each mask must be contiguous, that is, they must all be adjacent. You can’t have a blue bit sitting between two red bits. The standard RGB format is 0xaarrggbb where aa is the hex value of the alpha component, and rr, gg, and bb represent the hex values for the red, green, and blue components, respectively. This is represented in a direct color model as:

```
DirectColorModel rgbModel = new DirectColorModel(32,
0xff0000, 0x00ff00, 0x0000ff, 0xff000000)
```

The IndexColorModel Class

Unlike the DirectColorModel, the IndexColorModel class stores the actual red, green, blue, and alpha components of a pixel in a separate place from the pixel. A pixel value is an index into a table of colors. You can create an IndexColorModel by passing the number of bits per pixel, the number of entries in the table, and the red, green, and blue color components to the constructor.

You can optionally pass either the alpha components or the index value for the transparent pixel:

```
public IndexColorModel(int bitsPerPixel, int tableSize,
byte[] red, byte[] green, byte[] blue)
public IndexColorModel(int bitsPerPixel, int tableSize,
byte[] red, byte[] green, byte[] blue, int transparentPixel)
public IndexColorModel(int bitsPerPixel, int tableSize,
byte[] red, byte[] green, byte[] blue, byte[] alpha)
```

Instead of passing the red, green, and blue components in separate arrays, you can pass them as one big array of bytes. The IndexColorModel class assumes that every three bytes represents a color (every four if you tell it you are sending it alpha components). The color components should be stored in the order red, green, blue. If you specify an alpha component, it should come after the blue component. That might be counter-intuitive because the standard RGB format has the alpha component first. Here are the constructors for the packed format of colors:

```
public IndexColorModel(int bitsPerPixel, int tableSize,
```

byte[] packedTable, boolean includesAlpha)
public IndexColorModel(int bitsPerPixel, int tableSize,
byte[] packedTable, boolean includesAlpha, int transparentPixel)
Notice that you can actually have both a transparent pixel and alpha components using this last format!

You can retrieve a copy of the red, green, blue, and alpha tables with the following methods:

public final void getReds(byte[] redArray)
public final void getGreens(byte[] greenArray)
public final void getBlues(byte[] blueArray)
public final void getAlphas(byte[] alphaArray)

Each method copies the component values from the table into the array you pass it. Make sure that the array is at least as large as the table size. The getMapSize method returns the size of the table:

public final int getMapSize()

The getTransparentPixel method returns the index value of the transparent pixel, or it returns -1 if there is no transparent pixel:

public final int getTransparentPixel()
RGBImageFilter Class

The java.awt.image package comes with two standard image filters: the CropImageFilter and the RGBImageFilter. The RGBImageFilter enables you to manipulate the colors of an image without changing the image itself. When you create your own custom RGBImageFilter, you need to create only a filterRGB method:

public abstract int filterRGB(int x, int y, int rgb)

For each pixel in an image, the filterRGB method is passed the pixel’s x and y coordinates and its current RGB value. It returns the new RGB value for the pixel.

Because some images are defined with an index color model, you can set your filter to filter only the index color model. This is handy if the color adjustment has nothing to do with the x,y position of the pixel. If you filter only rgb values from the index, the x and y coordinates passed to filterRGB will be -1,-1. To indicate that you are willing to filter the index instead of the whole image, set the canFilterIndexColorModel variable to true:
protected boolean canFilterIndexColorModel

You can override the filterIndexColorModel method if you want to change the behavior of the index color model filtering:

```
public IndexColorModel filterIndexColorModel(IndexColorModel
oldCM)
```

The IndexColorModel returned by this method is the new index color model that will be used by the image.

If you want to change only the color model for an image, you can use the RGBImageFilter to substitute one color model for another:

```
public void substituteColorModel(ColorModel oldCM,
ColorModel newCM)
```

This method is used by the RGBImageFilter when filtering an index color model. It creates a new color model by filtering the colors of the old model through your filterRGB method and then sets up a substitution from the old color model to the new color model. When a substitution is set up, the filterRGB method is not called for individual pixels. This enables you to change the colors quickly.

Listing 5.20 shows a simple gray color model class that takes the red, green, and blue values from another color model and converts them all to gray. It takes the maximum value of the red, green, and blue components and uses it for all three components. The gray color model leaves the alpha value untouched.

Listing 5.20 Source Code for GrayModel.java

```
import java.awt.image.*;
// This class implements a gray color model
// scheme based on another color model. It acts
// like a gray filter. To compute the amount of
// gray for a pixel, it takes the max of the red,
// green, and blue components and uses that value
// for all three color components.
public class GrayModel extends ColorModel
{
ColorModel originalModel;
public GrayModel(ColorModel originalModel)
{
super(originalModel.getPixelSize());
this.originalModel = originalModel;
}
// The amount of gray is the max of the red, green, and blue
protected int getGrayLevel(int pixel)
{
```

```
return Math.max(originalModel.getRed(pixel),
Math.max(originalModel.getGreen(pixel),
originalModel.getBlue(pixel)));
}
// Leave the alpha values untouched
public int getAlpha(int pixel)
{
return originalModel.getAlpha(pixel);
}
// Since gray requires red, green and blue to be the same,
// use the same gray level value for red, green, and blue
public int getRed(int pixel)
{
return getGrayLevel(pixel);
}
public int getGreen(int pixel)
{
return getGrayLevel(pixel);
}
public int getBlue(int pixel)
{
return getGrayLevel(pixel);
}
// Normally, this method queries the red, green, blue and
// alpha values and returns them in the form 0xaarrggbb. To
// keep from computing the gray level 3 times, we just override
// this method, get the gray level once, and return it as the
// red, green, and blue, and add in the original alpha value.
public int getRGB(int pixel)
{
int gray = getGrayLevel(pixel);
return (getAlpha(pixel) << 24) + (gray << 16) +
(gray << 8) + gray;
}
}
```

Listing 5.21 shows an RGB image filter that sets up a simple substitution of the gray model for the original color model.

Listing 5.21 Source Code for GrayFilter.java

```
import java.awt.image.*;
// This class sets up a very simple image graying
// filter. It takes the original color model and
// sets up a substitition to a GrayModel.
public class GrayFilter extends RGBImageFilter
{
```

```
public GrayFilter()
{
canFilterIndexColorModel = true;
}
// When the color model is first set, create a gray
// model based on the original model and set it up as
// the substitute color model.
public void setColorModel(ColorModel cm)
{
substituteColorModel(cm, new GrayModel(cm));
}
// This method has to be present, but it will never be called
// because we are doing a color model substitution.
public int filterRGB(int x, int y, int pixel)
{
return pixel;
}
}
```

Listing 5.21 shows a simple applet that displays an image using the gray filter.

Listing 5.21 Source Code for Grayer.java

```
import java.awt.*;
import java.awt.image.*;
import java.applet.*;
// This applet displays a grayed-out image by using
// a GrayFilter rgb image filter.
public class Grayer extends Applet
{
private Image origImage;
private Image grayImage;
private GrayFilter colorFilter;
public synchronized void init()
{
// Get the name of the image to use
String gifName = getParameter(“image”);
// Fetch the image
origImage = getImage(getDocumentBase(), gifName);
System.out.println(origImage);
// Create the gray filter
colorFilter = new GrayFilter();
// Create a grayed-out version of the original image
grayImage = createImage(new FilteredImageSource(
origImage.getSource(),
colorFilter));
MediaTracker mt = new MediaTracker(this);
mt.addImage(grayImage, 0);
try {
mt.waitForAll();
} catch (Exception ignore) {
}
}
public synchronized void paint(Graphics g)
{
g.drawImage(grayImage, 0, 0, this);
}
public void update(Graphics g)
{
paint(g);
}
}
```

Animation by Color Cycling

The technique of color cycling is a little-known animation technique where an image is animated by changing its color palette without changing the actual image. This can take a number of forms—from simulating flowing water to changing text. You can use this technique on images created with an index color model. The idea is that you change the values in a color table and redraw the image with the new color table. If you continually loop through a set of colors, the image appears animated even though the image data itself hasn’t changed.

Any time you perform image animation by creating new images on-the-fly, don’t use createImage to create the new images. Instead, reuse the existing image by calling the flush method in the current image. This cleans out the memory used by the old image and causes it to be filtered again. Otherwise, on some systems you might use up more memory than you need to.

Listing 5.22 shows an RGB image filter that cycles the colors in an index color model.

Listing 5.22 Source Code for CycleFilter.java

```
import java.awt.*;
import java.awt.image.*;
//
// This class cycles the colors in an index color model.
// When you create a CycleFilter, you give the offset in the index color model and also the number of positions you want to cycle. Then every time you call cycleColors, it increments the cycle
```

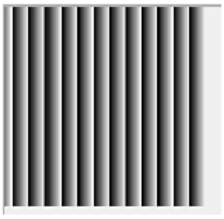
position. You then need to re-create your image and its colors will be cycled.This filter will work only on images that have an indexed color model.

```
public class CycleFilter extends RGBImageFilter {
// The offset in the index to begin cycling
protected int cycleStart;
// How many colors to cycle
protected int cycleLen;
// The current position in the cycle
protected int cyclePos;
// A temporary copy of the color components being cycled
protected byte[] tempComp;
public CycleFilter(int cycleStart, int cycleLen) {
this.cycleStart = cycleStart;
this.cycleLen = cycleLen;
tempComp = new byte[cycleLen];
cyclePos = 0;
// Must set this to true to allow the shortcut of filtering
// only the index and not each individual pixel
canFilterIndexColorModel = true;
}
// cycleColorComponent takes an array of bytes that represent
// either the red, green, blue, or alpha components from the
// index color model, and cycles them based on the cyclePos.
// It leaves the components that aren’t part of the cycle intact.
public void cycleColorComponent(byte component[]) {
// If there aren’t enough components to cycle, leave this alone
if (component.length < cycleStart + cycleLen) return;
// Make a temporary copy of the section to be cycled
System.arraycopy(component, cycleStart, tempComp,
0, cycleLen);
// Now for each position being cycled, shift the component over
// by cyclePos positions.
for (int i=0; i < cycleLen; i++) {
component[cycleStart+i] = tempComp[(cyclePos+i) %
cycleLen];
}
}
// cycleColors moves the cyclePos up by 1.
public void cycleColors() {
cyclePos = (cyclePos + 1) % cycleLen;
}
```

```
// Can’t really filter direct color model RGB this way, since we have
// no idea what rgb values get cycled, so just return the original
// rgb values.

public int filterRGB(int x, int y, int rgb) {
return rgb;
}
// filterIndexColorModel is called by the image filtering mechanism
// whenever the image uses an indexed color model and the
// canFilterIndexColorModel flag is set to true. This allows you
// to filter colors without filtering each and every pixel
// in the image.
public IndexColorModel filterIndexColorModel(IndexColorModel icm) {
// Get the size of the index color model
int mapSize = icm.getMapSize();
// Create space for the red, green, and blue components
byte reds[] = new byte[mapSize];
byte greens[] = new byte[mapSize];
byte blues[] = new byte[mapSize];
// Copy in the red components and cycle them
icm.getReds(reds);
cycleColorComponent(reds);
// Copy in the green components and cycle them
icm.getGreens(greens);
cycleColorComponent(greens);
// Copy in the blue components and cycle them
icm.getBlues(blues);
cycleColorComponent(blues);
// See if there is a transparent pixel. If not, copy in the alpha
// values, just in case the image should be partially transparent.
if (icm.getTransparentPixel() == -1) {
// Copy in the alpha components and cycle them
byte alphas[] = new byte[mapSize];
icm.getAlphas(alphas);
cycleColorComponent(alphas);
return new IndexColorModel(icm.getPixelSize(),
mapSize, reds, greens, blues, alphas);
} else {
// If there was a transparent pixel, ignore the alpha values and
// set the transparent pixel in the new filter
return new IndexColorModel(icm.getPixelSize(),
mapSize, reds, greens, blues,
icm.getTransparentPixel());
}
}
}
```

To use the CycleFilter, set up an applet that continually calls cycleColors in the CycleFilter and then redraws an image. Listing 5.23 shows an example applet that creates a simple memory image with an index color model and uses the CycleFilter to cycle the colors.

Figure 5.26 shows the output image generated by the Cycler applet.



Listing 5.23 Source Code for Cycler.java

```
import java.awt.*;
import java.awt.image.*;
import java.applet.*;
// This applet creates a series of moving
// lines by creating a memory image and cycling
// its color palette.
public class Cycler extends Applet implements Runnable {
protected Image origImage; // the image before color cycling
protected Image cycledImage; // image after cycling
protected CycleFilter colorFilter; // performs the cycling
protected Thread cycleThread;
protected int delay = 50; // milliseconds between cycles
protected int imageWidth = 200;
protected int imageHeight = 200;
protected boolean stopStatus = false; //thread should not stop, until true
public void init() {
// Create space for the memory image
byte pixels[] = new byte[imageWidth * imageHeight];
// We’re going to cycle through 16 colors, but leave position 0 alone in
// the index color model we create, so allow room for 17 slots
byte red[] = new byte[17];
byte green[] = new byte[17];
byte blue[] = new byte[17];
// Fill slots 1-16 with varying shades of gray (when the red,
green,
// blue values are all equal you get shades of gray ranging from
// black when all values are 0, to white when all values are 255).
for (int i=0; i < 16; i++) {
red[i+1] = (byte) (i * 16);
green[i+1] = (byte) (i * 16);
blue[i+1] = (byte) (i * 16);
}
// Create an index color model that supports 8 bit indices, only
17
// colors, and uses the red, green, and blue arrays for the color
values
IndexColorModel colorModel = new IndexColorModel(8, 17,
red, green, blue);
// Now create the image, just go from top to bottom, left to right
// filling in the colors from 1-16 and repeating.
for (int i=0; i < imageHeight; i++) {
for (int j=0; j < imageWidth; j++) {
pixels[i*imageWidth + j] =
(byte) ((j % 16)+1);
}
}
// Create the uncycled image
origImage = createImage(new
MemoryImageSource(imageWidth,
imageHeight,
colorModel, pixels, 0,
imageWidth));
// Create the filter for cycling the colors
colorFilter = new CycleFilter(1, 16);
// Create the first cycled image
cycledImage = createImage(new FilteredImageSource(
origImage.getSource(),
colorFilter));
}
// Paint simply draws the cycled image
public synchronized void paint(Graphics g) {
g.drawImage(cycledImage, 0, 0, this);
}
// Flicker-free update
public void update(Graphics g) {
```

```
paint(g);
}
// Cycles the colors and creates a new cycled image. Uses
media
// tracker to ensure that the new image has been created before
// trying to display. Otherwise, we can get bad flicker.
public synchronized void doCycle() {
// Cycle the colors
colorFilter.cycleColors();
// Flush clears out a loaded image without having to create a
// whole new one. When we use waitForID on this image now, it
// will be regenerated.
cycledImage.flush();
MediaTracker myTracker = new MediaTracker(this);
myTracker.addImage(cycledImage, 0);
try {
// Cause the cycledImage to be regenerated
if (!myTracker.waitForID(0, 1000)) {
return;
}
} catch (Exception ignore) {
}
// Now that we have reloaded the cycled image, ask that it
// be redrawn.
repaint();
}
// Typical threaded applet start and stop
public void start() {
stopStatus = false; //don’t stop yet
cycleThread = new Thread(this);
cycleThread.start();
}
public void stop() {
stopStatus = true;
}
public void run() {
// Continually cycle colors and wait.
while (!stopStatus) {
doCycle();
try {
Thread.sleep(delay);
} catch (Exception hell) {
}
}
```

```
}
}
```

When you are comfortable with Java’s imaging model, you can create many wonderful images. You can write image filters to perform a wide variety of effects. You can use the MemoryImageSource and PixelGrabber to make an image editor, or a paint program. You can even use image transparency to make interesting image combinations. Whatever image manipulation you need to do, Java should be able to handle it.

5.20 SUMMARY

We are covering Java Graphics, paint, Update, and repaint, Graphics Class ,Polygon Class. How to draw Text, different Drawing Modes , Drawing Images, MediaTracker Class, Graphics Utility Classes, Color Class, Clipping ,different Animation Techniques, Printing, Drawing Images to the screen. The concept of Producers, Consumers, and Observers are explained. Image Filtering, Copying Memory to an Image, Copying Images to Memory. Different Color Models, Graphics2D Object, Coordinates in Java 2D, Drawing Figures, Different Strokes, Custom Fills, Transformations, Drawing Text, Drawing Images, Transparency and Clipping concepts are explained in this chapter.

5.21 QUESTION

1 Explain the use of MediaTracker class.
2 What is the necessity of double-buffering?
3 What are the different animation techniques?
4 Explain Producers, Consumers, and Observers
5 What are the Different Color Models?
6 Explain Transparency and Clipping.

*****

6

JFC—JAVA FOUNDATION CLASSES

Unit Structure

6.1 INTODUCTION

When Java was first created, user interfaces were developed using the AWT classesHowever, the AWT design has many limitations. If you look at the buttons on the various machines, they actually appear different. Why is this?

Well, it's a concept called look and feel. The people who write software for the Macintosh are used to buttons looking a certain way, and Windows users are used to buttons looking a different way.

To reconcile this confusion, the designers of Java decided to use a design pattern that would give Java programmers access to the button but would use the system's own buttons for the look and feel. That means that when you put a java.awt.Button component on your screen, you're actually using a native button for the look and feel. So, on a Windows machine, a Windows button is created and on a Macintosh, a Macintosh button is created. Seems to make sense, right?

It’s also very difficult to create a truly abstract system and expect it to work the sameon all systems. Not all systems display characters the same, and from platform to platform individual characteristics of things such as TextAreas change. Because of all these variations, it’s very difficult to actually write an AWT system for one platform that will look and behave the same as an AWT system on another.

Because not all AWT systems are alike, people like you and me who are creating applications are forced to run a lot of tests on a lot of different types of computers in order to truly achieve Sun’s promise of Write Once Run Anywhere.

Unfortunately, all that testing can completely eat away all the shorter development time benefits of Java. So, several developers decided to build a different kind of system. The new systems relied on only one fairly common component—a Container (a parent of Panel). Because a Container is fairly uniform for all systems, you can paint (or draw) on one without encountering platform dependencies. Microsoft’s solution of this form was AFC (Application Foundation Classes), Netscape’s was IFC (Internet Foundation Classes), and there were a half dozen independentsolutions

6.2 JFC: A FIRST LOOK

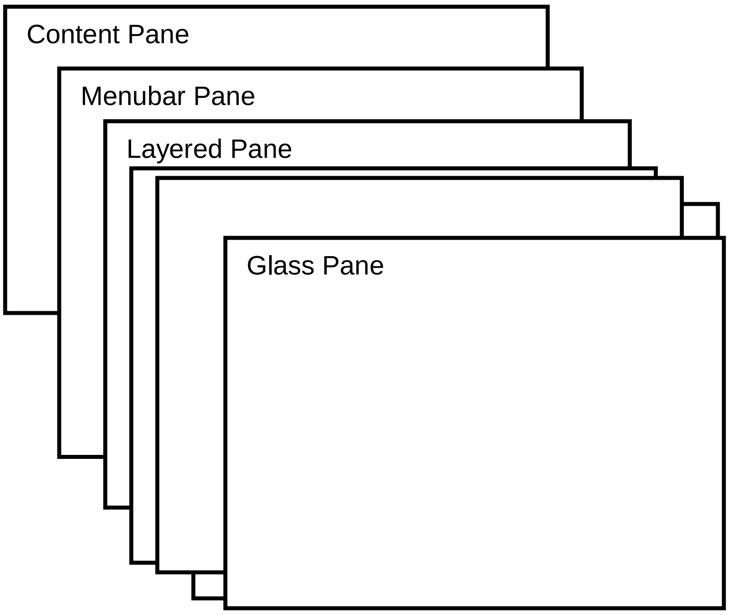
The JFC system is based on two primary things: first, the Container and Frame componentsfrom AWT and second, the JDK 1.1 event model. Unfortunately, because of the latter, IFC userswill find the switch to JFC much more difficult than users of AWT.Setting Up for JFCJFC is part of the core of JDK 1.2, but it can be added to JDK 1.1 implementationsHelloWorldwe will start our look at JFC by creating the simplest application possible. In Listing 6.1, you will find the source for HelloWorldJFC.java. After you compile and run HelloWorldJFC.
Listing 6.1 HelloWorldJFC.java—Hello World Written Using JFC

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
public class HelloWorldJFC extends JComponent
{
static JFrame myFrame;
public void paint(Graphics g)
{
g.setColor(Color.black);
g.drawString (“HelloWorld”,20,15);
}
public static void main(String args[])
{
myFrame = new JFrame(“Hello World!”);
HelloWorldJFC jt = new HelloWorldJFC();
myFrame.getContentPane().add(“Center”,jt);
myFrame.setSize(100,50);
myFrame.setVisible(true);
}
}
```

Pane Layering:

JFC uses multiple layers upon which it can layer components. This layering enables you tooverlap components and paint on top of components. Because the layering model is built intothe JFC system, unlike AWT, you can do this without getting inconsistent results. For instance,if you want to put a pop-up ToolTip on a button, and you want that ToolTip to appear directlybelow that button, even if something else is there, you can simply paint on the glass pane.Under AWT, this is not possible because when a button (or any component) occupies a placeon the screen, it generally does not allow you to paint over it.JFC includes a number of view layers; Figure 4 shows the panel views and their order.

Fig. 6.1



JFC layers severalpanes on top of oneanother.

All this means is that in our HelloWorld example’s main method, instead of simply adding thepanel, we add the panel to the content pane:

```
        myFrame.getContentPane().add(“Center”,jt);
```

Generally speaking, most of the time when you add a component to any JFC Container, you willadd it to the content pane. However, if the component has a specific need to overlay other components,you need to add it to either the layered pane or the glass pane.

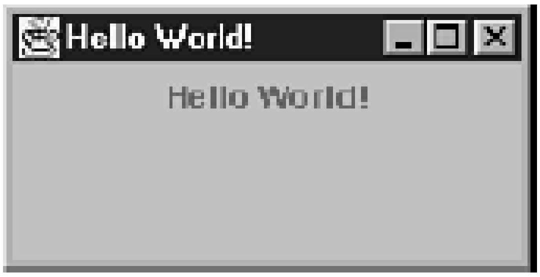
6.3 IMPROVING HELLOWORLD

Use of windowClosing() method to exit on pressing of Close button.The second thing for you to be concerned with is that instead of using a component (Label) todisplay the “Hello World” text, the program currently draws the string directly to the screen inthe paint() method. Although this is useful from the standpoint of an analogy with the applet’sHelloWorld, it’s not the bestpractice.

Listing 6.2 HelloWorldJFC2.java—Hello World Improved

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
public class HelloWorldJFC2 extends JPanel {
static JFrame myFrame;
public HelloWorldJFC2(){
JLabel label = new JLabel (“Hello World!”);
add(label);
}
public static void main(String args[]){
myFrame = new JFrame(“Hello World!”);
HelloWorldJFC2 hello = new HelloWorldJFC2();
myFrame.getContentPane().add(“Center”,hello);
myFrame.setSize(200,100);
myFrame.addWindowListener(new WindowAdapter() {
public void windowClosing(WindowEvent e) {System.exit(0);}
});
myFrame.setVisible(true);
}
}
```

Fig 6.2



JLabel :

JLabel can support not only text but also images (or both).we could have also added an icon that would have been displayed along with the text. There are two ways to do this:
-Use one of the constructors that accommodates the icon.
-Add the icon using the setIcon() method.

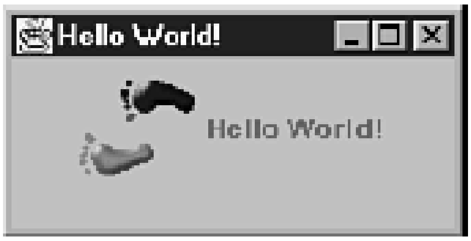
Adding Icons:

An icon is a graphical representation in JFC. The icon can be an image but might also be adrawing created programmatically. To add an icon to almost any component in the JFC set, wecan use one of the following two techniques: Specify the icon in the constructor.

Set the icon later using the setIcon() method.

```
public HelloWorldJFC2(){
Icon icon = new ImageIcon (“feet.gif”);
JLabel label = new JLabel (“Hello World!”, icon, SwingConstants.RIGHT);
add(label);
}
```

Fig 6.3



We can set the icon later on using setIcon().

```
public HelloWorldJFC2(){
Icon icon = new ImageIcon (“feet.gif”);
JLabel label = new JLabel (“Hello);
label.setIcon(icon);
add(label);
}
```

Closing the Window

To exit the application on pressing close button.

```
myFrame.addWindowListener(new WindowAdapter()
{
public void windowClosing(WindowEvent e)
{
System.exit(0);
}
})
```

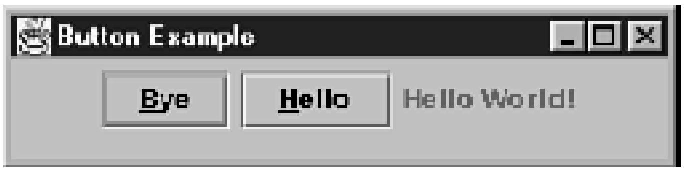
6.4 ADDING BUTTONS WITH JFC

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
public class ButtonExample extends JPanel {
static JFrame myFrame;
JLabel label;
public ButtonExample(){
label = new JLabel (“Hello World!”);
JButton hello = new JButton(“Hello”);
hello.setMnemonic(‘h’);
hello.addActionListener(new ActionListener(){
public void actionPerformed(ActionEvent ae){
```

continues

```
System.out.println(“Hello World!”);
label.setText(“Hello World!”);
}
});
JButton bye = new JButton(“Bye”);
bye.setMnemonic(‘b’);
bye.addActionListener(new ActionListener(){
public void actionPerformed(ActionEvent ae){
System.out.println(“Bye World!”);
label.setText(“Good Bye World!”);
}
});
add(bye);
add(hello);
add(label);
}
public static void main(String args[]){
myFrame = new JFrame(“Button Example”);
ButtonExample jt = new ButtonExample();
myFrame.getContentPane().add(“Center”,jt);
myFrame.setSize(300,70);
myFrame.addWindowListener(new WindowAdapter() {
public void windowClosing(WindowEvent e) {System.exit(0);}
});
myFrame.setVisible(true);
}
}
```

Fig 6.4



Setting a Shortcut Key or Mnemonic :

The mnemonic is the equivalent of the shortcut key.We can set shortcut key by setMnemonic() method: For the Following in combination with Alt key a shortcut is set.

```
hello.setMnemonic(‘h’);
```

Listening for Actions from the Button:

When a button is pressed, or its mnemonic is keyed, the button can produce an ActionEvent.To “hear” the action, add an ActionListener to the button and create an inner class that performsthe actions you want to occur when the button is pressed.

```
hello.addActionListener(new ActionListener(){
public void actionPerformed(ActionEvent ae){
System.out.println(“Hello World!”);
label.setText(“Hello World!”);
}
});
```

6.5ADDING TOOLTIPS AND ICONS

One of the advantages of having a layered view is that it affords JFC enhancements such asToolTips. ToolTips can be added to any JComponent, JButtons included.The label of a button is usually an abbreviation for what the button will actually do. This is thecase with the uttonExample in Listing 6.3.Also, like JLabel, JButtons can use icons as part of its label, so in Listing 6.4, you see theButtonExample modified to add the ToolTip to the buttons, and the feet.gif has been added tothe Hello button (see **Figure 6.5**).

Listing 6.4 TipButtons.java—Adding ToolTips to JFC Buttons

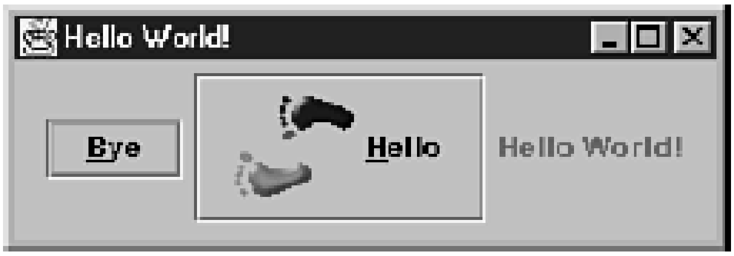
```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
public class TipButtons extends JPanel {
static JFrame myFrame;
protected JLabel label;
public TipButtons(){
```

```
label = new JLabel (“Hello World!”);
label.setOpaque(true);
```

continues

```
JButton hello = new JButton(“Hello”);
hello.setMnemonic(‘h’);
hello.addActionListener(new ActionListener(){
public void actionPerformed(ActionEvent ae){
label.setText(“Hello World!”);
}
});
//Set the ToolTip for the hello button
hello.setToolTipText(“Select to change label to Hello World”);
JButton bye = new JButton(“Bye”);
bye.setMnemonic(‘b’);
bye.addActionListener(new ActionListener(){
public void actionPerformed(ActionEvent ae){
label.setText(“Good Bye World!”);
}
});
//Set the ToolTip for the bye button
bye.setToolTipText(“Select to change label to Good Bye World”);
add(bye);
add(hello);
add(label);
}
public static void main(String args[]){
myFrame = new JFrame(“Tooltiped Buttons”);
TipButtons tb = new TipButtons();
myFrame.getContentPane().add(“Center”,tb);
myFrame.setSize(300,75);
myFrame.addWindowListener(new WindowAdapter() {
public void windowClosing(WindowEvent e) {System.exit(0);}
});
myFrame.setVisible(true);
}
}
```

Fig 6.5



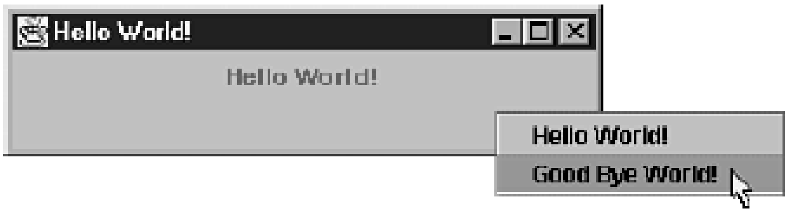
6.6 USING POP-UP MENUS

Like ToolTips, another one of the advantages of JFC’s design is an advanced pop-up menucapability. Often it’s useful to enable the user to click with the mouse and pop up an extramenu, as shown in **Figure 6.6**. Listing 6.5 shows just such an example.477

Listing 6.5 PopupExample.java—Adding Pop-Up Menus to a JFC Panel

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
public class PopupExample extends JPanel {
static JFrame myFrame;
protected JLabel label;
JPopupMenu popup;
public PopupExample(){
label = new JLabel (“Hello World!”);
label.setOpaque(true);
add(label);
popup = new JPopupMenu();
//create the first menu item
JMenuItem menuItem1 = new JMenuItem(“Hello World!”);
menuItem1.addActionListener(new ActionListener(){
public void actionPerformed(ActionEvent ae){
label.setText(“Hello World!”);
}
});
//create the second menu item
JMenuItem menuItem2 = new JMenuItem(“Good Bye World!”);
menuItem2.addActionListener(new ActionListener(){
public void actionPerformed(ActionEvent ae){
label.setText(“Good Bye World!”);
}
});
//add the menu items to the popup menu
popup.add(menuItem1);
popup.add(menuItem2);
addMouseListener(new MouseAdapter(){
public void mouseReleased(MouseEvent evt){
//Pop up the menu at the location where the mouse was pressed
if (evt.isPopupTrigger()){
popup.show(evt.getComponent(),evt.getX(),evt.getY());
}
}
});
}
public static void main(String args[]){
myFrame = new JFrame(“Popup Example”);
PopupExample example = new PopupExample();
myFrame.getContentPane().add(“Center”,example);
myFrame.setSize(300,75);
myFrame.addWindowListener(new WindowAdapter() {
continues
public void windowClosing(WindowEvent e) {System.exit(0);}
});
myFrame.setVisible(true);
}
}
```

Fig 6.6



Understanding PopupExample:

There are three primary steps to creating and showing a pop-up menu under JFC. The firststep is to create the pop-up menu itself.

```
popup = new JPopupMenu();
```

The next step is creating the menu items that will be on the pop-up. These menu items are thesame items that you add to a menu on a menu bar. In the case of PopupExample, you add twomenu items. Each of these menu items has an ActionListener added and created for it.

```
JMenuItem menuItem1 = new JMenuItem(“Hello World!”);
menuItem1.addActionListener(new ActionListener(){
public void actionPerformed(ActionEvent ae){
label.setText(“Hello World!”);
}
});
```

The final step is to actually pop up the menu. In this case, we want to pop up the menu wheneverthe pop-up trigger button is pressed (the right mouse button on most platforms).

TheprocessMouseEvent() method handles the work of popping up the menu.

```
addMouseListener(new MouseAdapter(){
public void mouseReleased(MouseEvent evt){
//Popup the menu at the location where the mouse was pressed
if (evt.isPopupTrigger()){
popup.show(evt.getComponent(),evt.getX(),evt.getY());
}
}
});
```

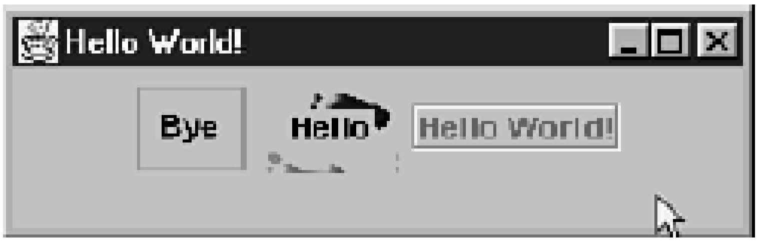
6.7 BORDERS

One of the unique characteristics that JFC has added to all components is the capability to havean adjustable border. If we want a button to have a flower border or a label to have an etched border,you can do so.
javax.swing.border package contains a number of different borders, eachof which can be applied to a wide variety of swing components.

Listing 6.6 shows the two buttons from Listing 6.3 with different borders (**see** Fig 6.7
).
Listing 6.6 BorderedButtons.java—Example with Bordered Components

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
import javax.swing.border.*;
public class BorderedButtons extends JPanel {
static JFrame myFrame;
protected JLabel label;
JPopupMenu pm;
public BorderedButtons(){
label = new JLabel (“Hello World!”);
label.setBorder(new EtchedBorder());
JButton hello = new JButton(“Hello”);
hello.addActionListener(new ActionListener(){
public void actionPerformed(ActionEvent ae){
label.setText(“Hello World!”);
}
});
//set the border to an image. Note: the image will be tiled.
Icon icon = new ImageIcon (“feet.gif”);
hello.setBorder(new MatteBorder(10, 10, 10, 10, icon));
JButton bye = new JButton(“Bye”);
bye.addActionListener(new ActionListener(){
public void actionPerformed(ActionEvent ae){
label.setText(“Good Bye World!”);
}
});
bye.setBackground (SystemColor.control);
bye.setBorder(new LineBorder(Color.green));
add(bye);
add(hello);
add(label);
}
public static void main(String args[]){
myFrame = new JFrame(“Border Example”);
BorderedButtons jt = new BorderedButtons();
myFrame.getContentPane().add(“Center”,jt);
myFrame.setSize(300,75);
myFrame.addWindowListener(new WindowAdapter() {
public void windowClosing(WindowEvent e) {System.exit(0);}
});
myFrame.setVisible(true);
}
```

Fig 6.7



Understanding BorderedButtons:

The key detailis that the label and each of the buttons have their own border. Each of them has a differentkind of border attached to it. In the case of the label, the etched border was used.label.setBorder(new EtchedBorder());

Any JComponent can have its border setusing the setBorder() method.

More Borders:

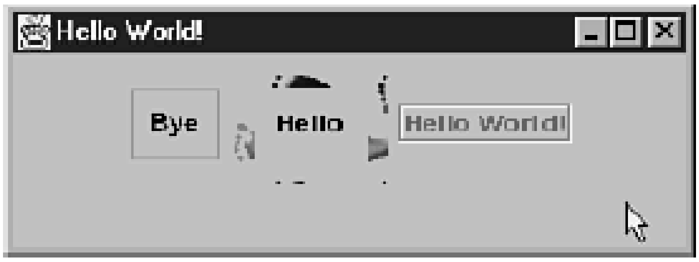
Borders can even be cascaded. As you saw in the BorderedButton example in Figure 6.9, the hello button uses the MatteBorder.

```
hello.setBorder(new MatteBorder(10, 10, 10, 10, icon));
```

The MatteBorder takes the icon image and tiles it for the border. The problem, however, is thatthe MatteBorder runs a bit too close to the text for most people.We can fix this problem by using the CompoundBorder. The CompoundBorder can cascadetwo borders together, like this:hello.setBorder(new CompoundBorder(new MatteBorder(10, 10, 10, 10, icon),new EmptyBorder(10,10,10,10)));
Buttons and labels with
borders.

FIG. 6.8

The BorderButtons withmore room around the border.

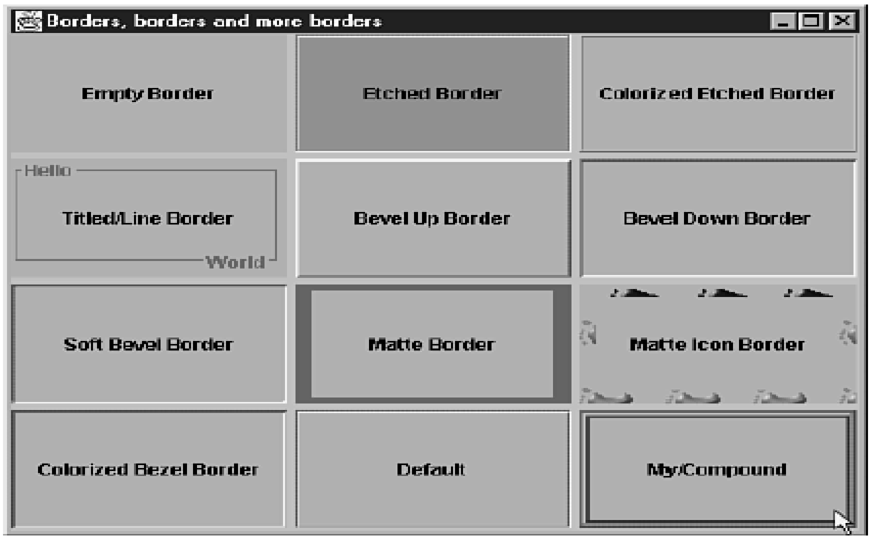


Some of the other borders also enable you to cascade them. In other words, you can combinethe capabilities of certain borders. Another good example of this is the TitledBorder. TheTitledBorder displays text within the border itself. So for instance, if you want to add the title“hi” to the border for the Hello button, you can change the set border lines:

```
LineBorder lb = new LineBorder(Color.green);
hello.setBorder(new TitledBorder(lb,”hi”));
```

There are quite a few borders in the borders package. Each of the borders has a slightly differentlook to it. Figure 6.9 shows several of the borders in the package.

Fig 6.9



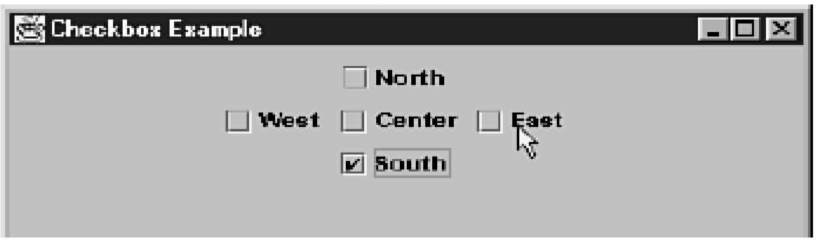
6.8 CHECK BOXES AND RADIO BUTTONS

Under JFC, check boxes and radio buttons are very similar. The only real difference betweenthe two is how they look. As with AWT, JFC check boxes and radio buttons come in twodifferentforms.

Listing 6.7 CheckBoxPanel.java—Using JFC Check Boxes

```
public class CheckBoxPanel extends JPanel implements SwingConstants{
public CheckBoxPanel(ActionListener al){
Box vertBox = Box.createVerticalBox();
Box topBox = Box.createHorizontalBox();
Box middleBox = Box.createHorizontalBox();
Box bottomBox = Box.createHorizontalBox();
ButtonGroup group = new ButtonGroup();
//Create the checkboxes
JCheckBox north = new JCheckBox(“North”);
north.addActionListener(al);
north.setActionCommand(“north”);
group.add(north);
topBox.add(north);
JCheckBox west = new JCheckBox(“West”);
west.addActionListener(al);
west.setActionCommand(“west”);
group.add(west);
middleBox.add(west);
JCheckBox center = new JCheckBox(“Center”);
center.addActionListener(al);
center.setActionCommand(“center”);
group.add(center);
middleBox.add(center);
JCheckBox east = new JCheckBox(“East”);
east.addActionListener(al);
east.setActionCommand(“east”);
group.add(east);
middleBox.add(east);
JCheckBox south = new JCheckBox(“South”);
south.addActionListener(al);
south.setActionCommand(“south”);
group.add(south);
bottomBox.add(south);
vertBox.add (topBox);
vertBox.add (middleBox);
vertBox.add (bottomBox);
add(vertBox);
}
}
```

Fig 6.10



6.9 APPLYING CHECKBOXPANEL TO CHANGE TEXT ALIGNMENT

We learned that when we use both anicon and a text label, we can specify the alignment of the icon to the text. We can change that alignment.JLabel and JButton both include two methods for specifying the position of the text, as it relatesto the icon:

```
setVerticalTextPosition()
setHorizontalTextPosition()
```

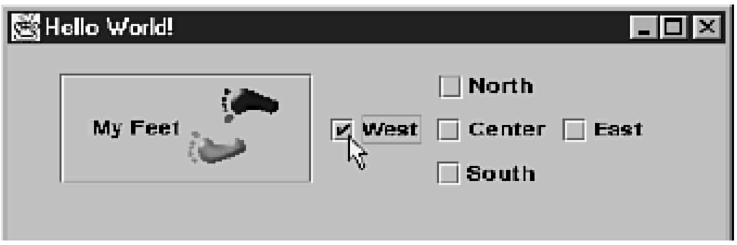
Both of these methods take a parameter from the SwingConstants interface. SwingConstantscontains a number of constant values such as: TOP, BOTTOM, LEFT, RIGHT, and CENTER. As youmight already have guessed, setting the vertical text position to say TOP causes the text toappear above the icon.

In Listing 6.7, you saw how a group of check boxes could be placed on a panel in a group.When each button is pressed, it creates an ActionEvent that gives a direction. Now use theCheckBoxPanel to change the alignment of the text and icon on a button, as shown in Listing 6.8 and Figure 6.11.

Listing 6.8 CheckBoxPanel—Altering Alignment

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
import javax.swing.border.*;
public class CheckBoxExample extends JPanel implements
 ActionListener,SwingConstants{
static JFrame myFrame;
protected JLabel label;
JButton theButton;
public CheckBoxExample(){
Icon icon = new ImageIcon (“feet.gif”);
theButton = new JButton(“My Feet”,icon);
add (theButton);
add (new CheckBoxPanel(this));
}
public void actionPerformed(ActionEvent ae){
String action = ae.getActionCommand();
if (action.equals(“north”)){
theButton.setVerticalTextPosition(TOP);
theButton.setHorizontalTextPosition(CENTER);
}
else if (action.equals(“south”)){
theButton.setVerticalTextPosition(BOTTOM);
theButton.setHorizontalTextPosition(CENTER);
}
else if (action.equals(“east”)){
theButton.setHorizontalTextPosition(RIGHT);
theButton.setVerticalTextPosition(CENTER);
}
else if (action.equals(“west”)){
theButton.setHorizontalTextPosition(LEFT);
theButton.setVerticalTextPosition(CENTER);
}
else if (action.equals(“center”)){
theButton.setHorizontalTextPosition(CENTER);
theButton.setVerticalTextPosition(CENTER);
}
}
public static void main(String args[]){
myFrame = new JFrame(“Checkbox Example”);
CheckBoxExample jt = new CheckBoxExample();
myFrame.getContentPane().add(“Center”,jt);
myFrame.setSize(400,250);
myFrame.addWindowListener(new WindowAdapter() {
public void windowClosing(WindowEvent e) {System.exit(0);}
});
myFrame.setVisible(true);
}
}
```

Fig 6.11



☐ North
☑ West ☐ Center ☐ East
☐ South

Changing the Alignment:

```
if (action.equals(“north”)){
theButton.setVerticalTextPosition(TOP);
theButton.setHorizontalTextPosition(CENTER);
}
```

6.10 TABBED PANES

Tabbed views have been a staple of GUI designs almost since the inception of the concept ofGUI. AWT includes a layout manager called CardLayout, which many people have used tocreate their own equivalent of a tabbed layout. However, AWT is not equipped with any ompletesolution for tabs. JFC has added a class called JTabbedPane for just this purpose.

The JTabbedPane automatically handles the graphical side of creating the tabs. LikeCardLayout, it also enables you to hide and show various pages each time you click on a tab.One of the great features of the graphic tabs themselves is that they can be placed at any of thefour standard sizes (top, bottom, left, or right). Listing 6.9 shows how to create two tabs at thetop.

Listing 6.9 JTabbedPane Provides a Facility to Handle Tabs in Your
Interfaces

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
public class TabExample extends JPanel {
static JFrame myFrame;
public TabExample(){
JTabbedPane tabs = new
JTabbedPane(SwingConstants.BOTTOM);
Icon icon = new ImageIcon (“feet.gif”);
JButton button = new JButton(icon);
JLabel label = new JLabel (“Hello World!”);
tabs.addTab(“Hello World”,label);
tabs.addTab(“Feet”,icon,button);
setLayout(new BorderLayout());
add(tabs,"Center");
}
public static void main(String args[]){
myFrame = new JFrame("Tab Example");
TabExample tabExample = new TabExample();
myFrame.getContentPane().add("Center",tabExample);
myFrame.setSize(400,200);
myFrame.addWindowListener(new WindowAdapter() {
public void windowClosing(WindowEvent e) {System.exit(0);}
});
myFrame.setVisible(true);
}
}
```

Understanding JTabbedPane :

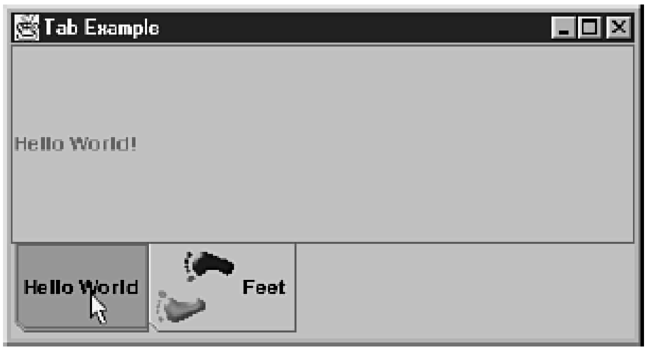
Fig 6.12 shows the results of running JTabbedPane. As with our other examples, the main()method should be easy for you to figure out on your own.In this example, two tabs have been added to the tabs object, using an addTab() method.There are three variations on the addTab() allowing you to label the tab with a string, an icon,or both. Here are the three methods:

addTab(String title, Component component);
addTab(String title, Icon icon, Component component);
addTab(String title, Icon icon, Component component, String toolTip);

Listing 6.9 shows the use of two of these options:

tabs.addTab("Hello World",label);
tabs.addTab("Feet",icon,button);

Fig 6.12



Other JTabbedPane Abilities:

if we choose to put an extra tab into the system at any time, itdoesn't necessarily have to be the last tab. To do so, you can use the insertTab() method:

public void insertTab(String title, Icon icon, Component component, String tip, int index)

6.11 SLIDERS

Sliders are similar to scrollbars in that they enable a user to drag a marker across the screen.

However, sliders are typically used to help specify a quantity, as opposed to moving a screenview. Unfortunately, sliders were missing among the components included with AWT. JFC,

however, has remedied this gap, with the JSlider class.JSlider can display major ticks, minor ticks, both, or neither as guides for the user. In addition,the slider can be displayed either vertically or horizontally. Listing 6.10 shows several differentvariations on the vertical variation of the slider (see Figure 6.13).

Listing 6.10 SliderExample.java—Five Different Sliders

```
import javax.swing.*;
import javax.swing.border.*;
import javax.swing.event.*;
import java.awt.BorderLayout;
import java.awt.event.WindowAdapter;
import java.awt.event.WindowEvent;
public class SliderExample extends JPanel{
JLabel slider5Value;
static JFrame myFrame;
public SliderExample() {
Box horizBox = Box.createHorizontalBox();
JSlider slider1 = new JSlider (JSlider.VERTICAL, 0, 50, 25);
slider1.setPaintTicks(true);
slider1.setMajorTickSpacing(10);
slider1.setMinorTickSpacing(2);
slider1.setSnapToTicks(true);
horizBox.add(slider1);
horizBox.add(horizBox.createHorizontalStrut(15));
JSlider slider2 = new JSlider (JSlider.VERTICAL, 0, 50,25);
slider2.setPaintTicks(true);
slider2.setMinorTickSpacing(5);
horizBox.add(slider2);
horizBox.add(horizBox.createHorizontalStrut(15));
JSlider slider3 = new JSlider (JSlider.VERTICAL, 0, 50,25);
slider3.setPaintTicks(true);
slider3.setMajorTickSpacing(10);
horizBox.add(slider3);
horizBox.add(horizBox.createHorizontalStrut(15));
JSlider slider4 = new JSlider (JSlider.VERTICAL, 0, 50,25);
slider4.setBorder(LineBorder.createBlackLineBorder());
horizBox.add(slider4);
horizBox.add(horizBox.createHorizontalStrut(15));
JSlider slider5 = new JSlider (JSlider.VERTICAL, 0, 50,25);
slider5.setBorder(LineBorder.createBlackLineBorder());
slider5.setMajorTickSpacing(10);
slider5.setPaintLabels(true);
horizBox.add(slider5);
horizBox.add(horizBox.createHorizontalStrut(15));
slider5Value = new JLabel("Slider5 value = 25");
horizBox.add(slider5Value);
slider5.addChangeListener(new ChangeListener(){
public void stateChanged(ChangeEvent event){
slider5Value.setText("Slider5 value = "
 +((JSlider)event.getSource()).getValue());
}
});
setLayout(new BorderLayout());
add(horizBox,"Center");
}
public static void main(String args[]){
myFrame = new JFrame("Slider Example");
SliderExample sliderExample = new SliderExample();
myFrame.getContentPane().add("Center",sliderExample);
myFrame.setSize(300,300);
myFrame.addWindowListener(new WindowAdapter() {
public void windowClosing(WindowEvent e) {System.exit(0);}
});
myFrame.setVisible(true);
}
}
```

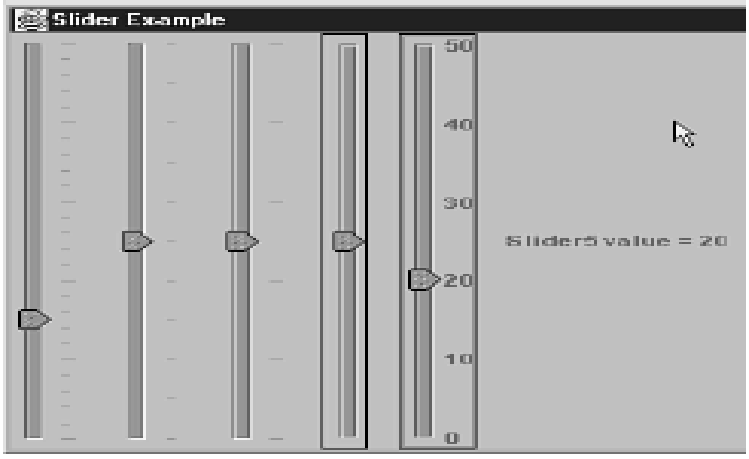
Understanding SliderExample

SliderExample uses the same box panel you saw in CheckBoxPanel. Each of the different sliders is added to this horizontal panel. Now, to create the various sliders, we used a constructor that takes four different values, like this:

JSlider slider1 = new JSlider (JSlider.VERTICAL, 0, 50, 25);

Output:

Fig 6.13



The first parameter for the constructor is obviously the direction, which can be eitherSwingConstants.VERTICAL or SwingConstants.HORIZONTAL. Note, that because JSlider implementsthe SwingConstants interface, we can also use the value like JSlider.VERTICAL.

The next two parameters for the constructor are the minimum and maximum values for theslider. So, can you guess what the last parameter is? It's the initial value of the slider.

Configuring the Tick Marks :

By default, a slider doesn't display its tick marks (like slider5). However, you can turn oneither the major or minor tick marks independently, using either setMajorTickSpacing() orsetMinorTickSpacing() respectively. Both of these methods take an int parameter. This parameterspecifies the number of elements between each tick.

Capturing Changes in the Slider:

To capture the changes in a slider, you can use a new event in the JFC set:

xxx.swing.event.ChangeEvent. The ChangeEvent occurs any time the slider is moved, so in

Listing 6.10, when slider5 is moved, the label at the far right is changed to include the valuefrom the slider.

6.12 PROGRESS BARS

When an activity is going to take a long time to complete, many applications use progress barsto show the current status, and to help the user know that the process is continuing. AWT doesnot include progress bars, but like sliders, JFC has filled this gap. The JProgressBar componentis JFC’s solution for the gap.

ProgressBarExample in Listing 6.11 demonstrates how a JProgressBar can be used. It createsa thread that progresses along and updates the bar, as shown in Figure.18.

```
import javax.swing.*;
import javax.swing.event.*;
import java.awt.*;
import java.awt.event.*;
public class ProgressExample extends JPanel {
ProgressThread progressThread;
JProgressBar progressBar;
static JFrame myFrame;
public ProgressExample() {
setLayout(new BorderLayout());
progressBar = new JProgressBar();
add(progressBar,”Center”);
JPanel buttonPanel = new JPanel();
JButton startButton = new JButton(“Start”);
buttonPanel.add(startButton);
startButton.addActionListener(new ActionListener() {
public void actionPerformed(ActionEvent e) {
startRunning();
}
});
JButton stopButton = new JButton(“Stop”);
buttonPanel.add(stopButton);
stopButton.addActionListener(new ActionListener() {
public void actionPerformed(ActionEvent e) {
stopRunning();
}
});
add(buttonPanel, BorderLayout.SOUTH);
}
public void startRunning() {
if(progressThread == null|| !progressThread.isAlive()) {
progressThread = new ProgressThread(progressBar);
```

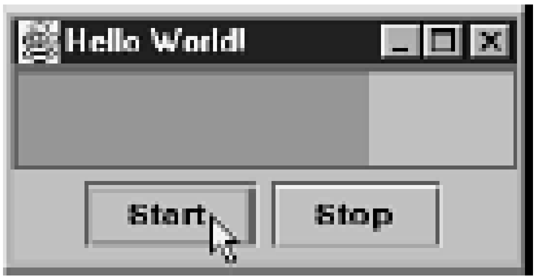
```
progressThread.start();
}
}
public void stopRunning() {
progressThread.setStop(true);
}
public static void main(String args[]){
myFrame = new JFrame(“Hello World!”);
ProgressExample progressExample = new ProgressExample();
myFrame.getContentPane().add(“Center”,progressExample);
myFrame.setSize(200,100);
myFrame.addWindowListener(new WindowAdapter() {
public void windowClosing(WindowEvent e) {
System.exit(0);
```

continues

```
}
});
myFrame.setVisible(true);
}
}
class ProgressThread extends Thread {
JProgressBar progressBar;
boolean stopStatus = false;
boolean aliveStatus = false;
public ProgressThread(JProgressBar progressBar){
this.progressBar = progressBar;
}
public void setStop(boolean value){
stopStatus = value;
}
public void run () {
int min = 0;
int max = 50;
progressBar.setMinimum(min);
progressBar.setMaximum(max);
progressBar.setValue(min);
for (int x=min;x<=max;x++) {
if(stopStatus){
break;
}else{
progressBar.setValue(x);
try {
```

```
Thread.sleep(100);
} catch (InterruptedException e) {
// Ignore Exceptions
}
}
}
aliveStatus = false;
}
}
```

Fig 6.14



UnderstandingProgressBarExampleCreating and Controlling the Progress Bar

To creating ad adding a progress bar is done like

```
progressBar = new JProgressBar();
add(progressBar,"Center");
```

The first set of method calls simply configures the initial state of the progress bar.

```
progressBar.setMinimum(min);
progressBar.setMaximum(max);
progressBar.setValue(min);
```

6.13 MENUS AND TOOLBARS

Menus and toolbars are common UI components in many systems. AWT provides menus, butleft out toolbars. JFC has provided both. The classes in question are JMenuBar and JToolBar.Both of them are standard JComponents. The interesting thing about this is that it means youcan put them into your UI in any location you would like. You are not bound to putting them inthe standard locations. So if you want a menu bar to be located below a text field, you can dojust that.You'll create a toolbar that will allow you to change the look-and-feel of the JTabbed pane in Listing 6.12, and Figure 6.6.The example adds a menu bar that has a File menu and an L&Fmenu. The File menu won't actually do anything at this point, but it does make it look nice.

Listing 6.12 Tab Example Can Be Expanded to Support Multiple Views

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
public class MenuBarExample extends JPanel implements
ItemListener{
static JFrame myFrame;
```

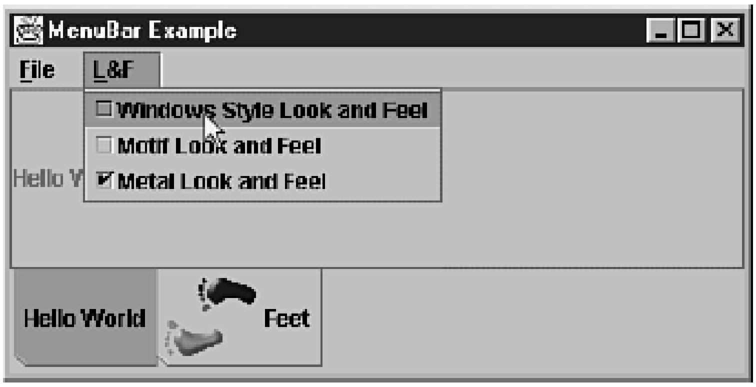
continues

```
Font myFont = new Font(“Dialog”, Font.PLAIN, 12);
public MenuBarExample(){
// setFont(myFont);
JTabbedPane tabs = new
JTabbedPane(SwingConstants.BOTTOM);
Icon icon = new ImageIcon (“feet.gif”);
JButton button = new JButton(icon);
JLabel label = new JLabel (“Hello World!”);
tabs.addTab(“Hello World”,label);
tabs.addTab(“Feet”,icon,button);
setLayout(new BorderLayout());
add(tabs,”Center”);
add(createMenu(),”North”);
}
public JMenuBar createMenu(){
JMenuBar menuBar = new JMenuBar();
// File Menu - create this so we have at least two menu options
JMenu file = (JMenu) menuBar.add(new JMenu(“File”));
file.setMnemonic(‘F’);
JMenuItem mi; //Temporary place holder
//Add several items under ‘File’ these won’t do anything.
mi = (JMenuItem) file.add(new JMenuItem(“Open”));
mi.setMnemonic(‘O’);
mi = (JMenuItem) file.add(new JMenuItem(“Save”));
mi.setMnemonic(‘S’);
mi = (JMenuItem) file.add(new JMenuItem(“Save As...”));
mi.setMnemonic(‘A’);
file.add(new JSeparator());
mi = (JMenuItem) file.add(new JMenuItem(“Exit”));
// Look and Feel Menu
JMenu options = (JMenu) menuBar.add(new JMenu(“L&F”));
options.setMnemonic(‘L’);
// Look and Feel Radio control
ButtonGroup group = new ButtonGroup();
```

```
mi = options.add(new JRadioButtonMenuItem(“Windows Style
Look and
Feel”));
mi.setActionCommand(“java.swing.plaf.windows.WindowsLookA
ndFeel”);
//If the current look and feel is windows, select this item.
mi.setSelected(UIManager.getLookAndFeel().getName().equals
 (“Windows”));
group.add(mi);
mi.addItemListener(this);
// mi.setAccelerator(KeyStroke.getKeyStroke
 (KeyEvent.VK_1, ActionEvent.ALT_MASK));
mi = options.add(new JRadioButtonMenuItem(“Motif Look and
Feel”));
mi.setActionCommand(“java.swing.plaf.motif.
 MotifLookAndFeel”);
mi.setSelected(UIManager.getLookAndFeel().getName().equals
 (“CDE/Motif”));
group.add(mi);
mi.addItemListener(this);
// mi.setAccelerator(
 KeyStroke.getKeyStroke(KeyEvent.VK_2,
ActionEvent.ALT_MASK));
mi = options.add(new JRadioButtonMenuItem(“Metal Look and
Feel”));
mi.setActionCommand(
 “java.swing.plaf.metal.MetalLookAndFeel”);
mi.setSelected(UIManager.getLookAndFeel().getName().equals
 (“Metal”));
// metalMenuItem.setSelected(true);
group.add(mi);
mi.addItemListener(this);
// metalMenuItem.setAccelerator(
 KeyStroke.getKeyStroke(KeyEvent.VK_3,
ActionEvent.ALT_MASK));
return menuBar;
}
public void itemStateChanged(ItemEvent e) {
Component root = myFrame;
//Bump the cursor into a wait mode while we make this change
root.setCursor(Cursor.getPredefinedCursor(Cursor.WAIT_CUR
SOR));
//Get the source of the event.
JRadioButtonMenuItem button = (JRadioButtonMenuItem)
e.getSource();
```

```
try {
if(button.isSelected()) {
UIManager.setLookAndFeel(button.getActionCommand());
button.setEnabled(true);
SwingUtilities.updateComponentTreeUI(myFrame);
}
} catch (UnsupportedLookAndFeelException exc) {
// Error - unsupported L&F
button.setEnabled(false);
System.err.println(“Unsupported LookAndFeel: “ +
 button.getText());
}catch (Exception exc2){
System.err.println(“Couldn’t load Look and feel” +
 button.getText());
}
root.setCursor(Cursor.getPredefinedCursor(Cursor.DEFAULT_
CURSOR));
}
public static void main(String args[]){
myFrame = new JFrame(“MenuBar Example”);
MenuBarExample menuExample = new MenuBarExample();
myFrame.getContentPane().add(“Center”,menuExample);
myFrame.setSize(400,200);
myFrame.addWindowListener(new WindowAdapter() {
public void windowClosing(WindowEvent e) {System.exit(0);}
});
myFrame.setVisible(true);
}
}
```

Fig 6.15



Understanding the MenuBar Example:

The two primary methods you’ll want to look at in this example are createMenu() and theitemStateChanged() event method. The createMenu() method produces the JMenuBar, whichis then added to the “North” portion of the JPanel.

createMenu() To understand the createMenu() method, it’s first important to understandthe basics of JMenuBar. A JMenuBar is really just a container that runs horizontally. It containsmenus that are the typical items you see on a menu bar, such as File and Edit. The menusthemselves are also containers that contain MenuItems. MenuItems are the actual items thatappear when you select a menu like Open or Save. Of course, if you’ve been building menubars with AWT, all of this should come as old hand.

Creating Menus The first task is for you to create the MenuBar. The menu bar will hold twomenus, File and L&F. To the menu bar you will add each of the JMenus. Now, the interestingpart comes when you start adding MenuItems. The JMenuItem works very similar to a JButton.It can have Mnemonics, ActionEvents and ItemEvents. Take a look at the first couple of lines ofcreateMenu(), in which the first menu (File) is created and its first item (Open) is added.

```
JMenu file = (JMenu) menuBar.add(new JMenu(“File”));
file.setMnemonic(‘F’);
JMenuItem mi; //Temporary place holder
mi = (JMenuItem) file.add(new JMenuItem(“Open”));
```

In addition to menu items, you might want to add a separation between items, as is the casebetween the Save As and the Exit.

```
file.add(new JSeparator());
```

Adding Check Box Menu Items Later in the method, you start adding the L&F menu. In thiscase, you don’t use the regular MenuItem class. Since you only want a single item to be selectedat a time, and also because it would be nice to have a visual cue to show which of the optionshas been selected, you use the JCheckBoxMenuItem. This class extends the normal JMenuItemclass and adds the capability to display both a check box and the label. This is the code for thefirst item:

```
ButtonGroup group = new ButtonGroup();
mi = lnf.add(new JCheckBoxMenuItem(“Windows Style Look and Feel”));
mi.setActionCommand(“java.swing.plaf.windows.
WindowsLookAndFeel”);
//If the current look and feel is windows, select this item.
mi.setSelected(UIManager.getLookAndFeel().getName().equals
(“Windows”));
group.add(mi);
```

Notice that the ButtonGroup class has been applied to make sure that only one of each of theoptions is selected at any time, by adding all the check box menu items to the group.

KeyAccelerators with KeyStroke :

You have already seen how the Mnemonic works with many JFC components. The menu itemsin Listing 6.12 each have a Mnemonic of their own. However, Mnemonics are triggered only whenthe components are visible. In the case of the MenuItem, it's convenient to use the Mnemonic sothat as soon as the pop-up menu is shown, the user can hot-key to that selection. But what ifyou want to be able to press a key combination at any time?

Because the pop-up is not visible all the time, the Mnemonic will not work. However, you can useanother trick to allow the user to press the key combination regardless of the menu's visibility.The trick is using a KeyAccelerator. The KeyAccelerator captures any keys that are pressedbut that are not used by some other device. (This means that if you have an active Mnemonicthat clashes with the accelerator, the Mnemonic will win.)

The setAccelerator()method is used for setting such a capture. setAccelerator() requiresa KeyStroke object as its parameter. KeyStroke is yet another new class in JFC, which is usedto represent a key combination, including a character and any key mask, such as Shift or Alt.So you can support a combination like Ctrl+b. The masks come from theawt.event.ActionEvent class. As you will recall, they are bit masks, and they include thefollowing:

```
java.awt.ActionEvent.SHIFT_MASK
java.awt.ActionEvent.CTRL_MASK
java.awt.ActionEvent.META_MASK
java.awt.ActionEvent.ALT_MASK
```

So if you want to capture Alt+1, you can set the menu item with this:

```
mi.setAccelerator(KeyStroke.getKeyStroke(KeyEvent.VK_1,
ActionEvent.ALT_MASK));
```

Because the masks are bit flags, you can combine them using bitwise ANDs (&). As an example,if you want to have a combination like Shift+Ctrl+B, you can do so by combining themasks like this:

```
mi.setAccelerator(KeyStroke.getKeyStroke(KeyEvent.VK_1,
ActionEvent.CTRL_MASK&ActionEvent.SHIFT_MASK));
```

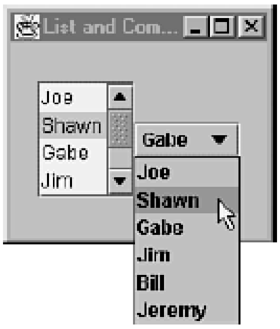
6.14 LISTS AND COMBO BOXES

Lists are probably one of the single-most-used UI components for displaying several items.Combo boxes are another variation on a list. AWT includes a list and a combo box (which isrepresented by the Choice class), but both are limited to displaying only strings. JFC addsJList, which is a completely different type of list box. JComboBox is different from AWT’sChoice in several ways, and it actually uses the JList class for its pop-up display. Instead ofLists and Combo Boxesdisplaying just strings, JList can display any kind of object and can include not only a stringbut also an associated icon. Further, the view for the list can change based on a large numberof conditions. So if you want to have a different view when the item is selected, it’s easy to do.First, you’ll create a simple list and combo box. Often you want to just have a list from a fixedlist. Listing 6.13 shows a list and a JComboBox, each with a fixed set of items (see Figure 6.8).

Listing 6.13 ListComboExample.java Shows a List of People

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
import javax.wing.border.*;
public class ListComboExample extends JPanel{
static JFrame myFrame;
String values[] = {“Joe”,”Shawn”,”Gabe”,”Jim”,”Bill”,”Jeremy”};
public ListComboExample(){
setLayout(new GridBagLayout());
JList list = new JList(values);
list.setVisibleRowCount(4);
JScrollPane pane = new JScrollPane();
pane.setViewportView(list);
add(pane);
JComboBox combobox = new JComboBox(values);
add(combobox);
}
public static void main(String args[]){
myFrame = new JFrame(“List and ComboBox Example”);
ListComboExample jt = new ListComboExample();
myFrame.getContentPane().add(“Center”,jt);
myFrame.setSize(400,250);
myFrame.addWindowListener(new WindowAdapter() {
public void windowClosing(WindowEvent e) {System.exit(0);}
});
myFrame.setVisible(true);
}
}
```

FIG. 6.16
A JList and JComboBox side by side.



Understanding ListComboExample :

By now, most of Listing 6.13 should be intuitive to you. The most interesting part of the list isthe use of the JScrollPane. JList doesn't implement scrolling on its own. It's an interestingdesign that allows you to do many interesting things with the JList without being bound tostandard scrolling. However, for most uses you will simply create the JScrollPane and set itsviewport view to the list. Notice that it's the pane, not the list, that is actually added to thepanel.

List View Models:

Although creating a list from a default list is useful, dynamic lists are far more interesting. Asyou will recall from the discussion of the Model-View-Controller earlier in this chapter, themodel provides the data for a component. So to use dynamic data with a list, it makes sensethat you will want to change the model. Lists use a basic list called a ListModel. ListModel isan interface that contains four methods, but it is often easier to simply extend theAbstractListModel class and define just two methods: getElementAt() and getSize().The combo box's model is slightly more complicated, however. First, it extends fromListModel, which means that you can base a ComboBoxModel on an existing ListModel, butafter that you must define two additional methods: setSelectedItem() andgetSelectedItem(). Listing 6.14 demonstrates how to modify Listing 6.13 to use modelsinstead of a fixed list of data.

Listing 6.14 Add ListModels and ComboBoxModels to Support Dynamic Data

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
import javax.swing.border.*;
public class ListComboExample2 extends JPanel{
```

```
static JFrame myFrame;
public ListComboExample2(){
setLayout(new GridLayout(2,2));
JList list = new JList(new ListModelExample());
list.setVisibleRowCount(4);
JScrollPane pane = new JScrollPane();
pane.setViewportView(list);
add(pane);
JComboBox combobox = new JComboBox(new
ComboModelExample());
add(combobox);
public static void main(String args[]){
myFrame = new JFrame(“List and ComboBox Example”);
ListComboExample2 jt = new ListComboExample2();
myFrame.getContentPane().add(“Center”,jt);
```

continues

Lists and Combo Boxes

```
myFrame.setSize(400,250);
myFrame.addWindowListener(new WindowAdapter() {
public void windowClosing(WindowEvent e) {System.exit(0);}
});
myFrame.setVisible(true);
}
class ListModelExample extends AbstractListModel{
String values[] = {“Joe”,”Shawn”,”Gabe”,”Jim”,”Bill”,”Jeremy”};
public Object getElementAt(int index){
return values[index];
}
public int getSize(){
return values.length;
}
}
class ComboModelExample extends ListModelExample
  implements ComboBoxModel{
Object item;
public void setSelectedItem(Object anItem){
item = anItem;
}
public Object getSelectedItem(){
return item;
}
}
}
```

6.15 USING TABLES

It has been argued that the invention of the spreadsheet was the single innovation that causedcomputers to come into the mainstream. One of the key features of a spreadsheet is the capabilityto display two-dimensional data in a table. Although not as functional as a full-fledgedspreadsheet, JFC has an extremely extensible table component called JTable.You’ll start by looking at the simplest version of a JTable, one in which you define a static set ofdata (Figure 6.9). Listing 6.15 shows just such an example, with a table that lists the firstseveral chapters in this book.

Listing 6.15 TableExample.java—JTable Allows You to Display Two-Dimensional Data

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
import javax.swing.border.*;
public class TableExample extends JPanel{
static JFrame myFrame;
String data[][] ={{“1”,”Introduction”},
{“2”,”What Java can Do for You”},{“3”,”JAVA Design”}
 ,{“4”,”Installing JAVA”},
{“5”,”JDK tools”},{“6”,”Object-Oriented Programming”}
 ,{“7”,”Hello world”},
{“8”,”Data Types”},{“9”,”Methods”},{“10”,”Using Expressions”}};
String columnNames[] ={“Chapter Number”,”Chapter Title”};
public TableExample(){
setLayout(new BorderLayout());
JTable table = new JTable(data,columnNames);
JScrollPane pane = JTable.createScrollPaneForTable(table);
add(pane);
}
public static void main(String args[]){
myFrame = new JFrame(“Table Example”);
TableExample tableExample = new TableExample();
myFrame.getContentPane().add(“Center”,tableExample);
myFrame.setSize(400,250);
myFrame.addWindowListener(new WindowAdapter() {
public void windowClosing(WindowEvent e) {System.exit(0);}
});
myFrame.setVisible(true);
}
}
```

FIG. 6.17 The JTable displays two-dimensional data elegantly.

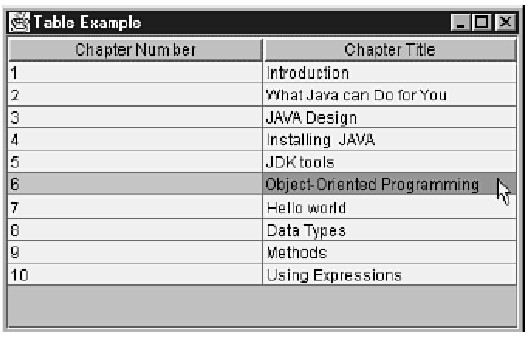


Table Example

Chapter Number	Chapter Title
1	Introduction
2	What Java can Do for You
3	JAVA Design
4	Installing JAVA
5	JDK tools
6	Object-Oriented Programming
7	Hello world
8	Data Types
9	Methods
10	Using Expressions

Understanding TableExample:

TableExample takes advantage of one of JTable's simplest constructors. The constructor takesa set of two-dimensional data, and a set of labels for the column heads. In TableExample, yousee the data and column heads represented as arrays of strings. The data is by no means limitedto strings, but for this example, that technique just was simplest. You can also use variouscomponents.Also, note that like JList, JTable does not implement its own scrolling. However, unlike JList,JTable has a convenience method for creating the scroll pane and placing the table and thetable header in the proper locations. Like JList, it's the pane, not the table itself, that gets added to the panel.

Table Models:

Although a complete coverage of all the JTable options is unfortunately beyond the scope ofthis book, this text would fall short if it did not talk about using the models for tables. You see,it's often not convenient to create the data for the table at the time you create it. Beyond that, itmight well be a problematic programming process to create the data as a two-dimensionalarray. However, the MVC model for the JTable is perfect for this situation. By defining themodel separately, the model can take on a life of its own without affecting the table code.Normally, you will create a table model by extending the AbstractTableModel class. TheAbstractTableModel has a few methods that need to be implemented:

npublic int getRowCount();

n public int getColumnCount();

npublic Object getValueAt(int row, int column);

You should be able to guess exactly how to implement these methods. Another table model,DefaultTableModel, extends AbstractTableModel but adds the capability to edit the table.

Listing 6.16 TableExample2.java—JTable Using a TableModel Instead of

Fixed Arrays

import java.awt.*;

```
import java.awt.event.*;
import javax.swing.*;
import javax.swing.table.*;
import javax.swing.event.*;
public class TableExample2 extends JPanel{
static JFrame myFrame;
public TableExample2(){
setLayout(new BorderLayout());
JTable table = new JTable(new TableModel());
JScrollPane pane = JTable.createScrollPaneForTable(table);
add(pane);
}
public static void main(String args[]){
myFrame = new JFrame(“Table Example #2”);
TableExample2 tableExample = new TableExample2();
myFrame.getContentPane().add(“Center”,tableExample);
myFrame.setSize(400,250);
myFrame.addWindowListener(new WindowAdapter() {
public void windowClosing(WindowEvent e) {System.exit(0);}
});
myFrame.setVisible(true);
}
}
class TableModel extends DefaultTableModel{
String data[][] ={{“1”,”Introduction”}
 ,{“2”,”What Java can Do for You”},{“3”,”JAVA Design”}
 ,{“4”,”Installing JAVA”},{“5”,”JDK tools”}
 ,{“6”,”Object-Oriented Programming”},{“7”,”Hello world”}
 ,{“8”,”Data Types”},{“9”,”Methods”},{“10”,”Using Expressions”}};
String columnNames[] ={“Chapter Number”,”Chapter Title”};
public int getRowCount(){
return data.length;
}
public int getColumnCount(){
return data[0].length;
}
public Object getValueAt(int row,int column){
return data[row][column];
}
public String getColumnName(int column){
return columnNames[column];
}
public void setValueAt(Object value, int row, int column){
```

```
if (value instanceof String){
data[row][column] = (String)value;
}
//Make sure you fire the table changed data so the view etc.
  will get notified of the change
fireTableChanged(new TableModelEvent(this,row,row,column));
}
}
```

Cell Editors:

When you run TableExample2, you will notice that if you try to type in one of the fields, itchanges to a text field and you are able to change the value. This can be very useful, but what ifyou want to offer the users a couple of options, like with a combo box? Well, tables support theconcept of a cell editor just for this purpose. The cell editor provides you incredibly finite controlover all aspects of the editing of a cell. For the scope of this chapter, though, the text willjust cover how to add two columns. One will use a combo box; the other, a check box. Listing 6.17 shows how to set one new column to be editable as a combo box. In addition, itdemonstrates the fact that a Boolean can be represented as a check box.

Listing 6.17 Adding a Separate Cell Editor to the TableExample

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
import javax.swing.table.*;
import javax.swing.event.*;
public class TableExample3 extends JPanel{
static JFrame myFrame;
public TableExample3(){
setLayout(new BorderLayout());
JTable table = new JTable(new TableModel());
JScrollPane pane = JTable.createScrollPaneForTable(table);
add(pane);
JComboBox comboBox = new JComboBox(new
String[]{“Low”,”Medium”,”High”});
TableColumn priorityColumn = table.getColumn(“Priority”);
priorityColumn.setCellEditor(new DefaultCellEditor(comboBox));
}
public static void main(String args[]){
myFrame = new JFrame(“Table Example #3”);
TableExample3 tableExample = new TableExample3();
myFrame.getContentPane().add(“Center”,tableExample);
myFrame.setSize(400,250);
myFrame.addWindowListener(new WindowAdapter() {
public void windowClosing(WindowEvent e) {System.exit(0);}
});
myFrame.setVisible(true);
}
}
class TableModel extends AbstractTableModel{
String data[][] ={{“1”,”Introduction”,”High”},
{“2”,”What Java can Do for You”,”Low”},
{“3”,”JAVA Design”,”Low”},{“4”,”Installing JAVA”,”Low”},
{“5”,”JDK tools”,”Low”},
{“6”,”Object-Oriented Programming”,”Low”},
{“7”,”Hello world”,”Low”},{“8”,”Data Types”,”Low”},
{“9”,”Methods”,”Low”},{“10”,”Using Expressions”,”Low”}};
Boolean doneFlags[] = {new Boolean(false),new Boolean(false),
new Boolean(false),new Boolean(false),new Boolean(false),
new Boolean(false),new Boolean(false),
new Boolean(false),new Boolean(false),new Boolean(false)};
String columnNames[] ={“Chapter Number”,”Chapter Title”,
”Priority”,”Done?”};
public int getRowCount(){
return data.length;
}
public int getColumnCount(){
return columnNames.length;
}
public Object getValueAt(int row,int column){
if (column<3)
return data[row][column];
else
return doneFlags[row];
}
public String getColumnName(int column){
return columnNames[column];
}
public void setValueAt(Object value, int row, int column){
if (value instanceof String){
data[row][column] = (String)value;
}else if (value instanceof Boolean){
doneFlags[row] =(Boolean) value;
}
fireTableChanged(new TableModelEvent(this,row,row,column));
}
```

```
public boolean isCellEditable(int row, int col) {return true;}
public Class getColumnClass(int c) {return getValueAt
 (0, c).getClass();}
}
```

6.16 TREES

Tree displays have been a huge hit with GUI designs for years. Trees allow you to display ndimensionaldata with relative ease. They are extremely popular for file Listing 6.s, but they areuseful in various situations. JFC's new tree class, called JTree, has many of the same facilitiesthat JTable and JList have for controlling data.Unlike with tables, it's not usual to create a tree from something as simple as an array ofstrings. If you want to do so, the facility is there, but what you will end up with is nothing morethan a glorified list. If you want to predefine the root headers, this method might be perfect foryou, but because it's not usual, we won't cover it here.

Tree Nodes:

Trees are made up of nodes. Each node represents either a leaf or a branch. A leaf is a distinctelement that has no sub elements or children. In other words, a leaf would not have the capabilityto "open" and "close" or, as it's put in tree terms, the capability to "expand" and "collapse." Abranch, on the other hand, does have children and can expand or collapse to show the children.The children can be either branches or leafs. In reality, both branches and leafs are standardnodes, and there is not much difference between the two, except that a leaf has no children.Generally speaking, you will probably find that the DefaultMutableTreeNode class will provideall the facilities you need for creating your tree nodes. Listing 6.18 shows how you could build atree using just DefaultMutableTreeNodes. Figure 6.10 shows the results of this code.

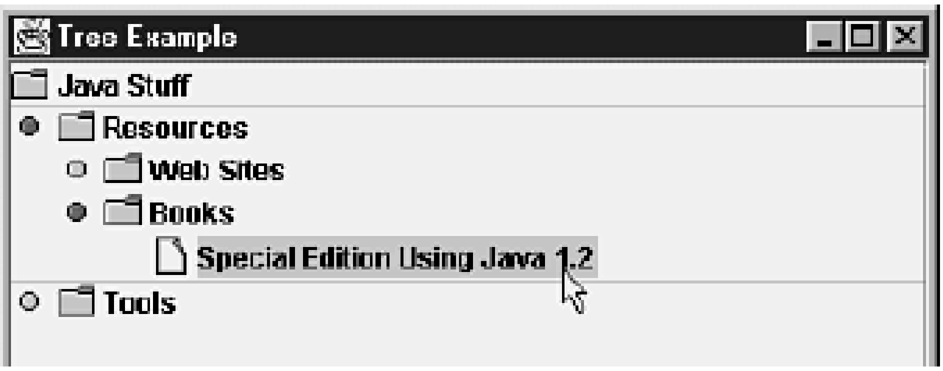
Listing 6.18 TreeExample.java—You Can Build a Tree from a Set of Tree Nodes

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
import javax.swing.tree.DefaultMutableTreeNode;
public class TreeExample extends JPanel{
static JFrame myFrame;
public TreeExample(){
setLayout(new BorderLayout());
DefaultMutableTreeNode rootNode = createNodes();
JTree tree = new JTree(rootNode);
tree.setRootVisible(true);
```

```
JScrollPane pane = new JScrollPane();
pane.setViewportView(tree);
add(pane);
}
public DefaultMutableTreeNode createNodes(){
DefaultMutableTreeNode rootNode =
  new DefaultMutableTreeNode (“Java Stuff”);
DefaultMutableTreeNode resources =
  new DefaultMutableTreeNode (“Resources”);
DefaultMutableTreeNode tools =
  new DefaultMutableTreeNode (“Tools”);
rootNode.add(resources);
rootNode.add(tools);
DefaultMutableTreeNode webSites =
  new DefaultMutableTreeNode (“Web Sites”);
DefaultMutableTreeNode books =
  new DefaultMutableTreeNode (“Books”);
resources.add(webSites);
resources.add(books);
DefaultMutableTreeNode magazines =
  new DefaultMutableTreeNode (“Magazines”);
webSites.add(new DefaultMutableTreeNode (“JavaSoft”));
webSites.add(new DefaultMutableTreeNode (“Gamelan”));
webSites.add(magazines);
magazines.add(new DefaultMutableTreeNode (“Javology”));
magazines.add(new DefaultMutableTreeNode (“JavaWorld”));
books.add(
  new DefaultMutableTreeNode (“Special Edition Using Java
1.2”));
tools.add(new DefaultMutableTreeNode (“JBuilder”));
tools.add(new DefaultMutableTreeNode (“Visual J++”));
tools.add(new DefaultMutableTreeNode (“Visual Age for Java”));
tools.add(new DefaultMutableTreeNode (“Apptivity”));
return rootNode;
}
public static void main(String args[]){
myFrame = new JFrame(“Tree Example”);
TreeExample treeExample = new TreeExample();
myFrame.getContentPane().add(“Center”,treeExample);
myFrame.setSize(400,250);
myFrame.addWindowListener(new WindowAdapter() {
public void windowClosing(WindowEvent e) {System.exit(0);}
});
myFrame.setVisible(true);
```

```
}
}
```

Fig 6.18



Understanding TreeExample:

First, take a look at the constructor. The tree is created usingthe rootNode object. You'll take a look at how the rootNode is created later, but the importantthing to realize is that you are passing in just the root node of the tree.The next line of code that you'll see sets the root node to be visible. This is actually the defaultvalue, but it's shown here to make the point. You see, the root node cannot have any siblings,and often this means that you will create a root node that doesn't really offer any additionalinformation. So you can set the root to be nonvisible by using tree.setRootVisible (false).The last interesting portion of the constructor is the fact that like JList and JTable, JTreedoes not handle the scrolling on its own. Instead, like JList and JTable, it implements theScrollable interface. So you create a scroll pane and add the tree to it.

Creating the Nodes The createNodes() method is where the nodes for the tree are actuallycreated. Effectively, what you are doing is layering out each branch. You first create therootNode, and then its children. Then the nodes for each of the children are created, and addedto the parent node, and so on.

Tree Models:

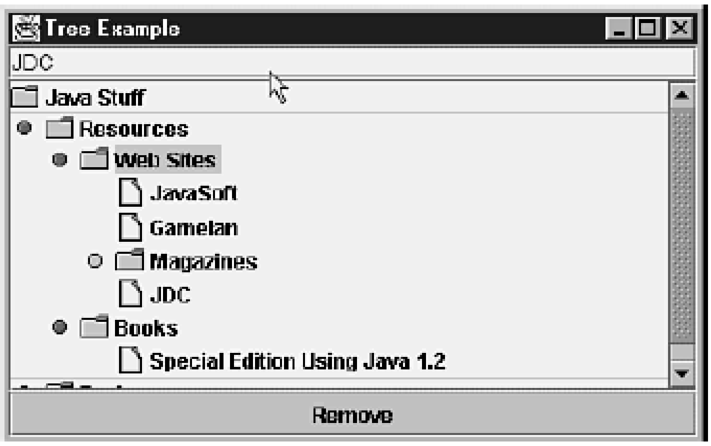
Building a tree out of a set of nodes as in Listing 6.18 works great when you know the data atthe time you create the tree. However, it's not enough when you will be inserting or removingnodes after you first create the tree. The reason is that although you can still call the add()method on a visible node, the tree doesn't know about this information, and the visual portionof the tree is not updated.Like the models for tables, the tree models can be manipulated at any time and can store variousvalues. All the details of handling TreeModels is unfortunately beyond the scope of thisbook, so we will cover only how to add and remove nodes

Listing 6.19 Trees Can Be Manipulated Through Their Models

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
import javax.swing.tree.DefaultMutableTreeNode;
import javax.swing.tree.DefaultTreeModel;
import javax.swing.tree.TreePath;
public class TreeExample extends JPanel{
static JFrame myFrame;
JTextField tf;
JTree tree;
public TreeExample(){
setLayout(new BorderLayout());
tf= new JTextField();
tf.addActionListener(new ActionListener(){
public void actionPerformed(ActionEvent ae){
addTextFieldValue();
}
});
add(tf,”North”);
DefaultMutableTreeNode rootNode = createNodes();
tree = new JTree(rootNode);
tree.setRootVisible(true);
JScrollPane pane = new JScrollPane();
pane.setViewportView(tree);
add(pane,”Center”);
JButton remove = new JButton(“Remove”);
remove.addActionListener(new ActionListener(){
public void actionPerformed(ActionEvent ae2){
removeSelectedNode();
}
});
add(remove,”South”);
}
public void removeSelectedNode(){
TreePath selectionPath = tree.getSelectionPath();
DefaultMutableTreeNode selectedNode =
(DefaultMutableTreeNode)
 selectionPath.getLastPathComponent();
((DefaultTreeModel)tree.getModel()).removeNodeFromParent
 (selectedNode);
}
public void addTextFieldValue(){
DefaultMutableTreeNode newNode = new
DefaultMutableTreeNode
```

```
  (tf.getText());
TreePath selectionPath = tree.getSelectionPath();
DefaultMutableTreeNode selectedNode =
(DefaultMutableTreeNode)
 selectionPath.getLastPathComponent();
((DefaultTreeModel)tree.getModel()).insertNodeInto(newNode,
  selectedNode, selectedNode.getChildCount());
}
public DefaultMutableTreeNode createNodes(){
DefaultMutableTreeNode rootNode = new
DefaultMutableTreeNode
  (“Java Stuff”);
DefaultMutableTreeNode resources = new
DefaultMutableTreeNode
  (“Resources”);
DefaultMutableTreeNode tools = new DefaultMutableTreeNode
  (“Tools”);
rootNode.add(resources);
rootNode.add(tools);
DefaultMutableTreeNode webSites =new
DefaultMutableTreeNode
  (“Web Sites”);
DefaultMutableTreeNode books = new DefaultMutableTreeNode
  (“Books”);
resources.add(webSites);
resources.add(books);
DefaultMutableTreeNode magazines = new
DefaultMutableTreeNode
  (“Magazines”);
webSites.add(new DefaultMutableTreeNode (“JavaSoft”));
webSites.add(new DefaultMutableTreeNode (“Gamelan”));
webSites.add(magazines);
magazines.add(new DefaultMutableTreeNode (“Javology”));
magazines.add(new DefaultMutableTreeNode (“JavaWorld”));
books.add(new DefaultMutableTreeNode
  (“Special Edition Using Java 1.2”));
tools.add(new DefaultMutableTreeNode (“JBuilder”));
tools.add(new DefaultMutableTreeNode (“Visual J++”));
tools.add(new DefaultMutableTreeNode (“Visual Age for Java”));
tools.add(new DefaultMutableTreeNode (“Apptivity”));
return rootNode;
}
public static void main(String args[]){
myFrame = new JFrame(“Tree Example”);
TreeExample treeExample = new TreeExample();
myFrame.getContentPane().add(“Center”,treeExample);
myFrame.setSize(400,250);
myFrame.addWindowListener(new WindowAdapter() {
public void windowClosing(WindowEvent e) {System.exit(0);}
});
myFrame.setVisible(true);
}
}
```

Fig 6.19



Understanding new TreeExample:

Most of this example is left for you to look through and figure out on your own. The two keymethods we need to explore, however, are removeSelectedNode() and addTextFieldValue().

removeSelectedNode()

The removeSelectedNode() method is designed to delete the node that has been selected inthe tree. To do this, it must first obtain the node that’s been selected. The tree can provide youwith the path to this node via the getSelectionPath() method. This method does not returnthe node itself, but instead returns a new class called a TreePath. The TreePath includes all thenodes from the root down to the node that has been selected. This is convenient because youcan store the path and later make sure that all the folders have been opened to expose the nodeusing tree’s expandPath() method, or it can be used in various other ways.In this case, the tree path is used to obtain the node that has been selected, but requesting thepath’s last component. GetLastPathComponent() actually returns an Object, so you will noticethat the result has been casted to a DefaultMutableTreeNode.Now that you know the node, you can delete it. As was mentioned earlier, the part of the treethat you want to have delete the node is the tree’s model. The model will then inform all theother necessary component pieces. Fortunately, DefaultTreeModel has a method just for thepurpose of removing the node called removeNodeFromParent that does the trick.

In this case, because you haven't used a special model and are using just the default, youcan safely cast the getModel() to a DefaultTreeModel. If you had changed it to acustom model, though, you'd need to adjust this code.You might also want to know that when you remove the node from the tree, the node itself is notactually deleted. If you wanted to put it somewhere else, or on another tree, you could safely do justthat.

addTextFieldValue()

The other method you're interested in with this class is the addTextFieldValue() method.This method creates a new node with the text in the text field and adds it as a child of the selectednode.

Using the TextField,you can add values tothe tree; using theremove button, you candelete from it.The first task in adding the node is to obtain its parent node. In the preceding section, youalready saw how most of the code for this task works. What's different is the last line of themethod. DefaultTreeModel's insertNodeInto() method takes three parameters. The first oneis the new node. The second is the new parent of the node, and the last is the location in the listto add the node. In this case, you will add it as the last child to the node, but you can also add itas the first element or in any other order you like. Play with this on your own to see the effect.

The Graphics2D Object:

One of the major complaints about the early versions of Java was that the graphics API was notvery robust. The functions provided by the AWT were roughly the same as those found on 8-bitmicrocomputers in the early 1980s. Java 1.2 introduces a powerful new 2D graphics API thatprovides the kinds of features one would expect from a modern graphics API. The Java 2D APIprovides better support for drawing shapes, filling and rotating shapes, drawing text, renderingimages, and defining colors.Under the old AWT API, we didn't actually create our own Graphics object (at least, notusually). Instead, we relied on the paint method, which received a Graphics object as a parameter.

Under the 2D API, we still use the paint method to get our Graphics object. UnderJava 1.2, the Graphics object passed to our paint method is really a Graphics2D object! Weonly need to cast the Graphics object to a Graphics2D object:

```
public void paint(Graphics oldGraphics)
{
Graphics2D newGraphics = (Graphics2D) oldGraphics;
// now you can use the new 2-D methods
}
```

Coordinates in Java 2D:

The original AWT API treated the drawing area as a simple field of pixels. Coordinate 1,1 representeda pixel location, and coordinate 2,1 was the pixel directly adjacent to 1,1. This drawingcoordinate system worked for simple screen drawing, but really didn't work well if you tried todraw on a printer. The problem is that printers have much greater resolutions than screens. Ifyou use the same number of pixels to draw an image on a printer as you do for the screen, theprinter image will be tiny (or very blocky).Java 2D has the notion of a "coordinate space." A *coordinate space* is just a way of specifyingcoordinates. Your program normally operates in "user coordinate space," which is a virtualdrawing area where coordinate 0,0 is in the upper-left corner, and the area is a certain numberof pixels wide and a certain number high.

Drawing Figures:

The Graphics2D object treats all drawn figures as shapes (it treats images and text separately).There is only one draw method in Graphics2D, and it takes a shape as a parameter:

```
public void draw(Shape s)
```

Likewise, there is only one fill method:

```
public void fill(Shape s)
```

Drawing a Line:

The java.awt.geom.Line2D class is a *shape* that represents a simple line between two points.When you create the line, we must use either the Float or Double version of the line class todefine the points. The Double version is there in case you need a lot of precision. The followingcode snippet creates a Line2D object from 50,60 to 300,320 and then draws it:

```
Line2D line = new Line2D.Float(50, 60, 300, 320);
newGraphics.draw(line);
```

We can also use the Point2D class to define the end points of a line:

```
Line2D line = new Line2D.Float(new Point2D.Float(50, 60), new
Point2D.Float(300, 320));
```

Drawing a Rectangle:

The Rectangle2D class defines a rectangle shape using an x,y coordinate, a width, and a height. The following code snippet creates a rectangle at 10,15 with a width of 100 and a height of 50 and draws it:

```
Rectangle2D rect = new Rectangle2D.Float( 10, 15, 100, 50);
newGraphics.draw(rect);
```

Drawing a Rounded Rectangle:

The RoundRectangle2D draws a rectangle whose corners are rounded. In addition to the rectangleposition and size, we specify the width and height of the rounding arc. If we specify awidth and height of 5 for the rounding arc, for example, the rounded portion of each cornerwould be 5 units in width and height. The following code fragment defines a rounded rectangleat 10, 15 with a width of 100, a height of 50, and a rounding of 5 units in the x direction, and 3 inthe y direction:

```
RoundRectangle2D roundRect =New
RoundRectangle2D.Float(10, 15, 100, 50, 5, 3);
```

Drawing Ellipses and Circles :

The Ellipse2D represents an ellipse shape, which is a circle when the width and height are thesame. As with the ellipse methods in the Graphics object, the ellipse is defined using abounding box. We specify a rectangle whose width and height represent the width and heightof the ellipse (for a circle, use the circle’s diameter for both width and height). You also give the x,y coordinate of the upper-left corner of the rectangle. The following code fragment createsan ellipse at coordinates 50, 80 with a width of 30 and a height of 10:

```
Ellipse2D ellipse =New Ellipse2D.Float(50, 80, 30, 10);
```

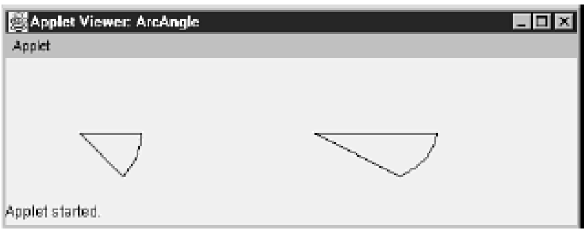
Drawing Arcs:

In addition to drawing ellipses, you can draw partial ellipses using the Arc2D class.

Drawing Curves:

In addition to arcs, the Java 2D API provides shapes that define quadratic and cubic curves. A quadratic curve is defined by two end points and a single control point that controls the shape of the curve. A cubic curve is similar to a quadratic curve except that it has two control points rather than one.

Fig 6.20



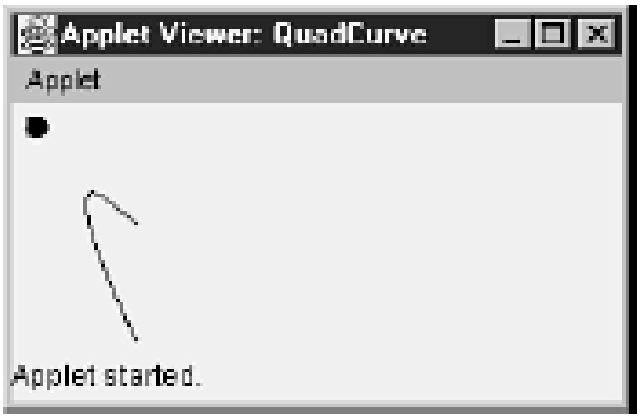
Applet started.

The following code fragment defines a quadratic curve with end points at 50, 50 and 50, 200with a control point at 10, 10:

```
QuadCurve2D curve = new QuadCurve2D.Float(50, 50, 10, 10, 50, 100);
```

Figure 6.21 shows the curve with a circle drawn at the control point.

FIG. 6.21



Applet started.

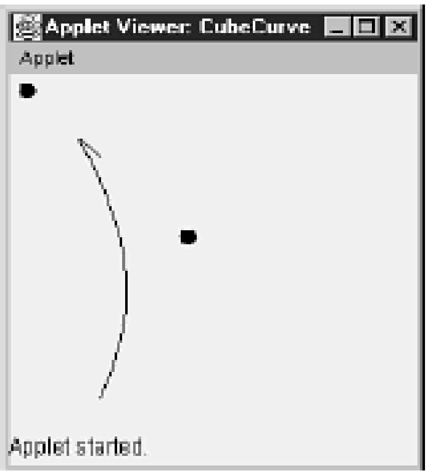
The following code fragment defines a cubic curve with end points at 50, 50 and 50, 200, withcontrol points at 10, 10 and 100, 100:

```
CubicCurve2D curve = new CubicCurve2D.Float(50, 50, 10, 10, 100, 100, 50, 200);
```

Figure 6.22 shows the cubic curve with circles drawn at the control points.

FIG. 6.22

A quadratic curve bendstoward the control point.



Applet started.

Drawing Arbitrary Shapes :

In addition to predefined shapes, we can also create our own shapes using the GeneralPathclass. All we need to do to create our own shape is call the moveTo and lineTo methods inGeneralPath for each point in the shape, just as if you were drawing it with the Graphics class.The following code fragment creates a GeneralPath object and defines a triangle using moveToand lineTo:

```
GeneralPath path = new GeneralPath();
path.moveTo(0, 50);
path.lineTo(25, 0);
path.lineTo(50, 50);
```

In addition to the lineTo method, we can also create a curve between two points using quadToand curveTo. The quadTo method starts at the current point and draws a quadratic curve toanother point using a single control point. The following code defines a quadratic line to 50,50using 100,0 as a control point : path.quadTo(100, 0, 50, 50);

The curveTo method uses two control points, as shown in this statement, which is the same asthe preceding quadTo call but with an additional control point of 40,30:

```
path.curveTo(40, 30, 100, 0, 50, 50);
```

Different Strokes:

The drawing stroke defines how the border of the shape is drawn. In the old AWT Graphics object, the stroke was always a 1-pixel wide solid line.Now, we can change the width of the stroke and also create a wide variety of dotted lines.

The BasicStroke class can be created several different ways:

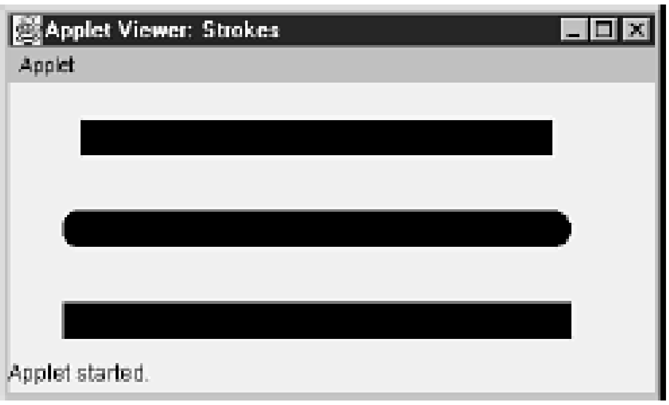
```
public BasicStroke()
public BasicStroke(float width, int cap, int join)
public BasicStroke(float width, int cap, int join,
float miterlimit)
public BasicStroke(float width, int cap,
int join, float miterlimit, float[] dash,float dash_phase)
```

The width parameter defines the width of the stroke. The default width is 1. The cap parameterdetermines the shape of the ends of the stroke. It is important for line segments and curvesthat are not connected all the way around. There are three possible values for the cap parameter:

CAP_BUTT, CAP_ROUND, and CAP_SQUARE.

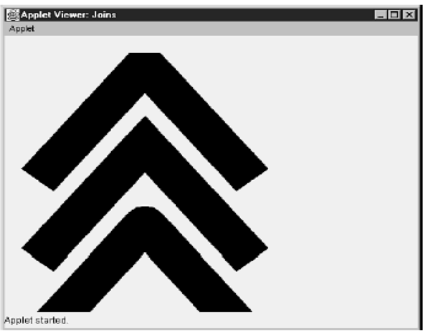
The CAP_BUTT value indicates that there should be nothing extra drawn on the end of thestroke. CAP_ROUND indicates that the ends of the line should be rounded, and CAP_SQUARE indicatesthat the ends should be square. Figure 26.4 shows the various ends of a stroke.

Fig 6.23



The join parameter determines how corners in the stroke should be handled. There are threepossible values for the join parameter: JOIN_BEVEL, JOIN_MITER, and JOIN_ROUND.The JOIN_BEVEL option draws a straight line between the outer ends of the stroke and fills inthe area. This gives the corners a flattened appearance. The JOIN_MITER option extends thestrokes until they meet at a point, making the corners sharp. The JOIN_ROUND option makes anarc connection between the ends of the stroke, giving the corner a rounded appearance. Themiterlimit parameter defines how far out the miter can go. It is not used for JOIN_ROUND orJOIN_BEVEL. Figure 6.23 shows the various types of corner joins.

Fig 6.24

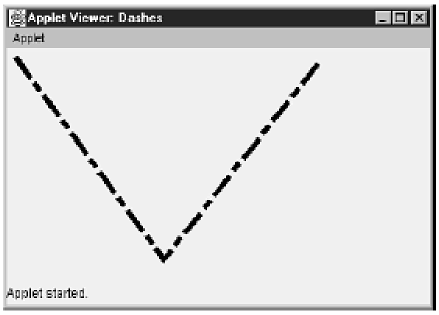


The dash array contains alternating lengths for blank space and dashes for a line. The firstelement of the array indicates the length of the first dash. The next element indicates thelength of the first gap. The array values continue to alternate between dash length and gaplength. The following array definition, for example, creates a dash-dot pattern by specifying along dash, a small gap, an even smaller second dash, and another gap the same size as the first:

```
float[] dashValues = new float[4];
dashValues[0] = 20;
dashValues[1] = 10;
dashValues[2] = 5;
dashValues[3] = 10;
```

Figure 6.25 shows this dashed line pattern.

Fig 6.25



The dashphase parameter tells how far into the dash sequence to start. If you used a value of 10for the first dash length, and you set the dash phase to 11, for example, the line would startwith the first blank area.

After you have created a BasicStroke object, you tell the Graphics2D class to use it by callingsetStroke:

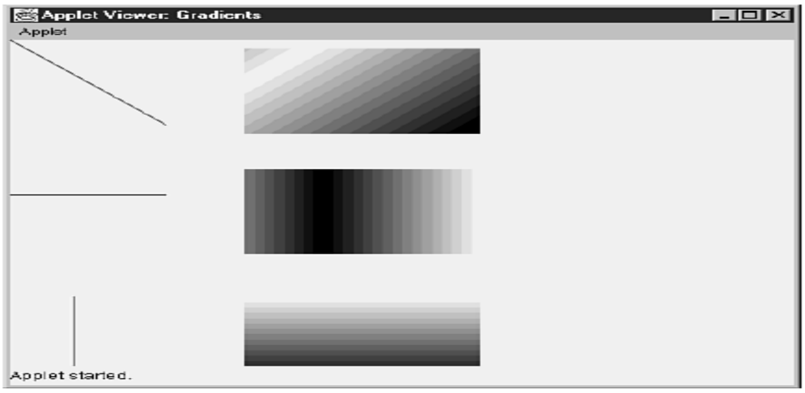
```
newGraphics.setStroke(myStroke);
```

Custom Fills :

Figure 6.25 shows three gradient patterns and lines indicating the endpoints of the gradient.
The gradient sounds pretty complex, but it is very easy to set one up. The GradientPaint
object has the following Constructors:

```
public GradientPaint(float x1, float y1, Color color1,float x2, float y2, Color color2)
public GradientPaint(float x1, float y1, Color color1,float x2, float y2, Color color2, boolean cyclic)
public GradientPaint(Point2D pt1, Color color1,Point2D pt2, Color color2)
public GradientPaint(Point2D pt1, Color color1,Point2D pt2, Color color2, boolean cyclic)
```

Fig 6.26



After we create the GradientPaint object, call the setPaint method in Graphics2D to use it:

```
GradientPaint gradient = new GradientPaint(0, 0, Color.red,50, 50, Color.blue, true);

newGraphics.setPaint(gradient);
Rectangle2D rect = new Rectangle2D.float(0, 0, 200, 200);
NewGraphics.fill(rect);
```

The TexturePaint object enables you to use an image for filling figures. You just provide a
BufferedImage object containing the pattern you want to use for the fill.The TexturePaint object has only one Constructor:
public TexturePaint(BufferedImage pattern,Rectangle2D rect2d, int interpolation)

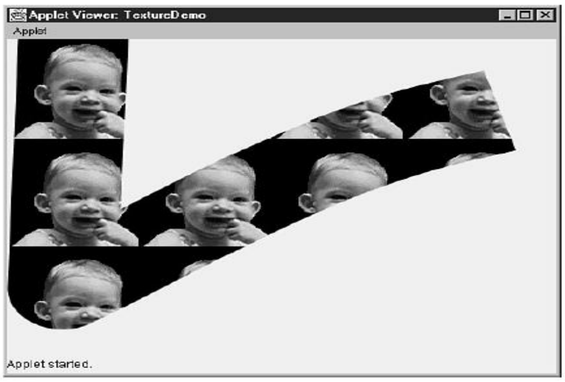
The pattern parameter specifies the image used for the fill pattern. The rect2d parameterdescribes the size of the pattern. In other words, the pattern is replicated in blocks the size ofrect2d. The interpolation parameter is used to determine how to display colors when the fillcan’t represent all the colors in the image. The two possible values for interpolation areTexturePaint.BILINEAR and TexturePaint.NEAREST_NEIGHBOR.

Listing 6.20 Source Code for TextureDemo.java

```
import java.awt.*;
import java.applet.*;
import java.awt.geom.*;
import java.awt.image.BufferedImage;
import java.net.URL;
public class TextureDemo extends Applet
{
public void paint(Graphics g)
{
// Get the Graphics2D object
```

```
Graphics2D newG = (Graphics2D) g;
// Create a path to draw
GeneralPath path = new GeneralPath();
path.moveTo(60, 0);
path.lineTo(50, 300);
path.curveTo(160, 230, 270, 140, 400, 100);
// Load an image to use as the texture
URL imgURL = null;
try {
imgURL = new URL(getDocumentBase(), “katyface.gif”);
} catch (Exception ignore) {
}
Image img = getImage(imgURL);
MediaTracker tracker = new MediaTracker(this);
try {
tracker.addImage(img, 0);
tracker.waitForAll();
} catch (Exception e) {
e.printStackTrace();
}
// Normally you would create a buffered image and set the individual
// pixels yourself, but you can also use an existing image. Rather than
// trying to convert the loaded image into a BufferedImage, it is easier
// (from a programming standpoint) to just get a Graphics2D object for
// the BufferedImage and draw the texture image into it
BufferedImage buff = new BufferedImage(img.getWidth(this),
img.getHeight(this), BufferedImage.TYPE_INT_RGB);
Graphics tempGr = buff.createGraphics();
tempGr.drawImage(img, 0, 0, this);
// The TexturePaint requires a rectangle defining the area to be filled
// In this case, just use the image size.
Rectangle2D rect = new Rectangle2D.Float(
0, 0, img.getHeight(this),
img.getWidth(this));
// Create the textured paint
TexturePaint painter = new TexturePaint(buff, rect,
TexturePaint.NEAREST_NEIGHBOR);
newG.setPaint(painter);
```

```
// Create a VERY wide stroke (100 pixels) round off the corners and
// make the ends square
BasicStroke stroke = new BasicStroke(100,
BasicStroke.CAP_SQUARE, BasicStroke.JOIN_ROUND);
newG.setStroke(stroke);
// Draw the original path
newG.draw(path);
}
}
```



Transformations :

In addition to drawing shapes, you frequently need to move, rotate, and resize shapes. Theseoperations are performed using an object called AffineTransform. The AffineTransform classcontains a matrix that is used to change one x,y point into another.The formula for translating point xorig,yorig to xnew,ynew is as follows:

xnew = xorig * m00 + yorig * m01 + m02
ynew = xorig * m10 + yorig * m11 + m12

the AffineTransform class has methods for building the transform one operation at a time. We start by creating an emptyAffineTransform by calling the default Constructor:

```
AffineTransform myTransform = new AffineTransform();
```

We can add a translation (that is, a move) by calling the translate method. The followingexample moves the transform 5 units to the left and 10 units down:

```
myTransform.translate(-5.0, 10.0);
```

The rotate method rotates the transform in a clockwise direction (or counter-clockwise fornegative angle values). The angle of rotation is specified in radians. We can specify an optionalx,y point, which rotates the transform around a specific point rather than around the origin.

The two forms of the rotate method are as follows:

```
public void rotate(double numRadians)
public void rotate(double numRadians, double x, double y)
```

The scale method multiplies the x and y coordinates by particular values. When you call thescale method, you always pass scale values for both x and y. If you only want to scale in onedirection, use a scale value of 1.0 for the value you don't want to change. To double the size ofthe transform in the x direction and leave the y values alone, for example, use 2.0 for the xscale and 1.0 for the y scale:myTransform.scale(2.0, 1.0);

The shear method is similar to the scale method, except that it changes the x value based onthe y value, and the y value based on the x value. If you shear the x values by a factor of 2, the xvalue for each point will be increased by the shear factor times that point's y value. Suppose, forexample, that you have an x shear of 2, the point 5,10 would be sheared to (5 + 2*10), 10 or insimpler terms, 25, 10. The higher the y value, the more the x points get moved. The samerelationship holds true for shears in the y direction. The shear method is defined as follows:

```
public void shear(double xshear, double yshear)
```

Just manipulating an AffineTransform object doesn't change anything onscreen. You need toapply the transform to either a shape that you want to draw, or to the entire Graphics2D object.

To manipulate a shape, call the createTransformedShape method, which returns a new shapewith the current transform applied. The following code fragment rotates a rectangle 45 degrees,

for example:

```
Rectangle2D rect = new Rectangle2D.Float(0, 0, 50, 20);
AffineTransform transform = new AffineTransform();
Transform.rotate(45.0 * 3.1415927 / 180.0);
Shape rotatedRect = transform.createTransformedShape(rect);
```

We can also apply a transform to the entire Graphics2D object by calling the setTransformmethod in Graphics2D. Transforming the Graphics object itself causes everything drawn withthat Graphics object to be transformed before being displayed. The setTransform method isdefined as follows:

```
public void setTransform(AffineTransform transform)
```

Drawing Text :

Drawing text is one of the big three in graphics operations (drawing shapes and images beingthe other two). The 2D API adds some handy extensions to the original text drawing routines.One of the most notable is the ability to rotate text. The Graphics2D class implements severaldifferent versions of the drawString method. The simplest version takes a string and an x,ycoordinate pair:public void drawString(String textString, float x, float y)

Styled Strings :

We can also draw a StyledString object, which is a string and an associated set of attributes(often just a font). We can even concatenate styled strings together to make a single styles string containing text of different fonts. We might want to print the ubiquitous “Hello World!”message, for example, using different fonts for each word. The following code fragment does just that:Font fntCourier = new Font(“courier”, Font.PLAIN, 48);

```
StyledString ssHello = new StyledString(“Hello “, fntCourier);
Font fntHelvetica = new Font(“helvetica”, Font.BOLD, 48);
StyledString ssWorld = new StyledString(“World!”, fntHelvetica);
StyledString helloWorld = ssHello.concat(ssWorld);
newGraphics.drawString(helloWorld, 100, 100);
```

Like the normal Java String class, the StyledString is immutable—we can’t change its contents.
Instead, we create new versions of the string by calling concat (to put two strings together)
or substring (to get a portion of the string). The substring method is identical to the

String substring method:

```
public StyledString substring(int startIndex, int endIndex)
```

Text Layouts:

The TextLayout class has several Constructors:

```
public TextLayout(String string, Font font)
public TextLayout(String string, AttributeSet attributes)
public TextLayout(StyledString text)
public TextLayout(AttributedCharacterIterator text)
```

The two most common TextLayout constructors are the String/Font combination and theStyledString. If you want more complex characters, you can use a set of character attributes, or define the attributes of each character though an iterator. The program in Listing 6.21 displays a simple paragraph onscreen:

Listing 6.21 Source Code for Paragraph.java

```
import java.awt.*;
import java.applet.*;
import java.awt.geom.*;
import java.awt.font.*;
public class Paragraph extends Applet
```

```
{
public void paint(Graphics g)
{
// Get the Graphics2D object
Graphics2D newG = (Graphics2D) g;
String message = “Harold, on the other hand, refused “+
“ to eat the chalk. He reached into his lunchbox “+
“ and removed a small block of balsa wood, which “+
“ he proceeded to chew on gleefully. Harold always “+
“ referred to balsa wood as a \”light snack\””;
Font fntRoman = new Font(“timesroman”, Font.PLAIN, 24);
TextLayout layout = new TextLayout(message, fntRoman);
newG.drawString(layout, (float) 100, (float) 100);
}
}
```

Character Attributes

The names of the valid text attributes are as follows:

Attribute Name	**Meaning**
LANGUAGE	The language used for the text (usually a locale)
READING	The pronunciation information for the word (used for some languages that require a pronunciation annotation)
INPUT_METHOD_SEGMENT	Used for breaking up lines into segments (usually words)
SWAP_COLORS	Whether the foreground and background colors of the text should be swapped
FAMILY	The family for the font
WEIGHT	The weight of the characters (bold text means a heavier weight)
POSTURE	The slant of the text
SIZE	The size of the font in points
TRANSFORM	The graphics transform applied to the font
FONT	An instance of Font to use for the characters
BIDI_EMBEDDING	Controls bi-directional text
BACKGROUND	An instance of Color specifying the background color of the text
FOREGROUND	An instance of Color specifying the foreground color of the text
UNDERLINE	Indicates whether text should be underlined

STRIKETHROUGH	Indicates whether the text should have a strikethrough
SUPERSUBSCRIPT	Makes the text either a superscript or a subscript
JUSTIFICATION	Adjusts the amount of space used in justification
RUN_DIRECTION	Controls whether text runs left-to-right, right-toleft, or top-to-bottom
BIDI_NUMERIC	Controls bi-directional layout of roman numerals
BASELINE	Adjusts the baseline for all characters

To set various attributes, you just call the add method with the name of the attribute and its

value. Many attribute values have preset constants; others take string or numeric arguments.

The following code fragment creates a text attribute set with a Times-Roman font andstrikethrough:

```
Font fntRoman = new Font(“timesroman”, Font.PLAIN, 24);
TextAttributeSet textAttr = new TextAttributeSet();
textAttr.add(TextAttributeSet.FONT, fntRoman);
textAttr.add(TextAttributeSet.STRIKETHROUGH,
TextAttributeSet.STRIKETHROUGH_ON);
```

Drawing Images:

The Java 2D API recognizes the frequent need to manipulate images by performing variousmathematical operations on them. It also strengthens the ability to manipulate image data on apixel-by-pixel basis. Although the earlier versions of the AWT did not prevent these types ofoperations, they were not as efficient, and many operations needed to be performed manually.

Buffered Images:

In previous versions of Java, it was very difficult to manipulate images on a pixel-by-pixel basis.

You had to either create an image filter and modify the pixels as they came through the filter,or you had to make a pixel grabber to grab an image and then create a MemoryImageSource toturn the array of pixels into an image. The BufferedImage class provides a quick, convenient shortcut by providing an image whose pixels can be manipulated directly.

The easiest way to create a buffered image is to specify a width, height, and a pixel type:public BufferedImage(int width, int height, int pixelType)

The pixel type has many different options, but you usually just need TYPE_INT_ARGB orTYPE_INT_RGB. The TYPE_INT_ARGB pixel type is the same pixel type used in earlier versions ofJava. Each pixel is represented by a 32-bit integer with 8 bits for transparency, and 8-bit red,green, and blue values, arranged as aarrggbb. The TYPE_INT_RGB format is almost identicalexcept that it assumes that there is no transparency. The aa portion is assumed to be 255 at alltimes. If you have worked with systems that encode colors in bbggrr form in an integer, youcan use the TYPE_INT_BGR format for your pixels. Given three integer red, green, and bluevalues, you can encode them in RGB format like this:

```
int rgb = (red << 16) + (green << 8) + blue;
```

To use this technique, you must ensure that the red, green, and blue values are between 0 and255. To add an alpha (transparency) to the pixel, you can use the following line:

```
int argb = (alpha << 24) + (red << 16) + (green << 8) + blue;
```

To store a pixel in bgr format, just reverse the order of the variables like this:

```
int bgr = (blue << 16) + (green << 8) + red;
```

To extract the alpha, red, green, and blue components from an ARGB pixel, you can use thefollowing statements:

```
int alpha = (argb >> 24) & 255;
int red = (argb >> 16) & 255;
int green = (argb >> 8) & 255;
int blue = argb & 255;
```

The getRGB method in BufferedImage returns a pixel in ARGB format, regardless of how the pixel is stored within the image:

```
public int getRGB(int x, int y)
```

To set a pixel in a buffered image, call setRGB (don’t forget to use ARGB format for the pixel):

```
public int setRGB(int x, int y, int rgb)
```

Copying an Image into a BufferedImage:

The following code fragment creates a BufferedImage object, and then gets a Graphics objectto draw to the buffered image, and then draws an existing image into the buffered image:

```
// Create a buffered image using the existing image’s
// width and height
BufferedImage buff = new BufferedImage(img.getWidth(this),
img.getHeight(this), BufferedImage.TYPE_INT_RGB);
// Get a graphics object for drawing into the buffered image
Graphics tempGr = buff.createGraphics();
// Draw the existing image into the buffered image
tempGr.drawImage(img, 0, 0, this);
```

Filtering Buffered Images:

To filter an image using any of the BufferedImageOp classes, just create the op and call the filter method with a source and destination image (which for some ops can be the same image). Here is an example call to the filter method:

```
op.filter(srcBuff, destBuff);
```

If you just need to draw a filtered image, the drawImage method in Graphics2D enables you todraw an image filtered by a BufferedImageOp:

```
public void drawImage(BufferedImage image,BufferedImageOp op,
 ImageObserver obs)
```

AffineTransformOp An AffineTransformOp performs an affine transform on an image. Thetransform can contain any combination of scaling, rotation, and translation. The following codefragment creates an AffineTransformOp that rotates an image 45 degrees, for example:

```
AffineTransform transform = new AffineTransform();
transform.rotate(45.0 * 3.1415927 / 180.0,
buff.getWidth() / 2, buff.getHeight() / 2);
AffineTransformOp op = new BilinearAffineTransformOp(transform);
```

BandCombineOp At first glance, the BandCombineOp filter seems strange and not very useful. Buffered images are stored in raster objects, which are really just arrays of pixels. Within these rasters are “bands” of colors. For a typical RGB image, there are three bands: red, green, and blue. The BandCombineOp filter enables you to change the color of an image by a combination of the various color bands. For a 3-band image, you specify a matrix with 4 columns and 3 rows, like this:

B11 B12 B13 B1OFFSET

B21 B22 B23 B2OFFSET
B31 B32 B33 B3OFFSET

The formula for the color of band 1 is as follows:
B1COLOR = B1COLOR * B11 + B2COLOR * B12 +
B3COLOR * B13 + B1OFFSET

The following filter would do nothing to an image (that is, it is the identity matrix for a band combine) because it just multiplies each color in the band by 1:

1.0 0.0 0.0 0.0
0.0 1.0 0.0 0.0
0.0 0.0 1.0 0.0

In a typical RGB raster, band 1 is red, band 2 is green, and band 3 is blue. The following matrixremoves all the green color from a picture while leaving red and blue alone:

1.0 0.0 0.0 0.0
0.0 0.0 0.0 0.0
0.0 0.0 1.0 0.0

You can invert colors, too. To invert the red, for instance, the red row in the matrix would bethis:

-1.0 0.0 0.0 255.0

This would multiply the red color by –1 and add 255, making the formula 255 – red.

The really interesting combinations come when you allow one color to contribute to anothercolor. You could filter the red so that it is a combination of the amount of red, green, and bluein the image, for example. The following matrix leaves red and blue alone, but for green it useshalf the old amount of green and one fourth the amount of red and one fourth the amount ofblue. In other words, the brighter the green and blue are, the brighter the red is. Here is thematrix:

1.0 0.0 0.0 0.0
0.25 0.5 0.25 0.0
0.0 0.0 1.0 0.0

The following code fragment creates a BandCombineOp object:

```
float filt[][] = {
{ 1.0f, 0.0f, 0.0f, 0.0f },
{ 0.25f, 0.5f, 0.25f, 0.0f },
{ 0.0f, 0.0f, 1.0f, 0.0f }};
```

BandCombineOp op = new BandCombineOp(filt);

BandCombineOp can use the same image for the source and destination.

ColorConvertOp The ColorConvertOp class converts from one color space to another. A color

space defines how a color is represented. You are probably familiar with the RGB color space,

for example, where you specify colors by using the amount of red, green, and blue in the color.

Publishers often use a color space called CMYK, which specifies colors by the amount of cyan,

magenta, yellow, and black. The following code fragment creates a ColorConvertOp that convertsan image into a grayscale image:

ColorSpace graySpace = ColorSpace.getInstance(ColorSpace.CS_GRAY);

ColorConvertOp op = new ColorConvertOp(graySpace);

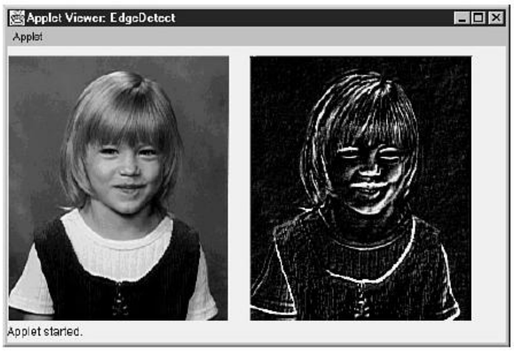
ConvolveOp The ConvolveOp class implements a common image operation where each pixelis modified based on the pixels around it, according to a simple matrix operation. When you Drawing Imagescreate a ConvolveOp object, you supply a Kernel object, which contains the matrix to be appliedto the image. The Kernel object even has several predefined sharpening matrices that useedge-detection to enhance an image.

If you have ever used a paint program that has different image algorithms, you may have seenone called “edge detect.” When you run edge detect on an image, you get a mostly black imagewith lines showing some of the edges in the image. You can create an edge detect using aKernel and a ConvoleOp.When you create a Kernel for image processing, you really just specify a matrix that is used tocalculate the color of each pixel in the image. The matrix determines how the surroundingpixels affect the current pixel. In the case of an edge detect, you completely ignore the color ofthe current pixel. Instead, you look at the upper-left and lower-right pixels (you can really lookat any pair of opposing pixels). Multiply the upper-left pixel by some factor and multiply thelower-right pixel by the negative of that factor. The higher the factors, the more pronouncedthe edges.

The matrix for the Kernel object should have odd-numbered dimensions, and is usually 3*3.

The center value in the matrix is the multiplication factor for the current pixel. The surroundingvalues are the factors for the surrounding pixels. To perform an edge detect, you want thecurrent pixel to be ignored, so the center value would be 0. The upper-left and lower-rightcorners of the matrix would contain the edge-detect factors. Figure 26.9 shows an image nextto the result of an edge detect on the image.

Fig 6.27



The following code fragment creates a fairly strong edge detector using a factor of 5:

```
float matrix[] = {
-5, 0, 0,
0, 0, 0,
0, 0, 5 };
Kernel kernel = new Kernel(3, 3, matrix);
ConvolveOp op = new ConvolveOp(kernel,
ConvolveOp.EDGE_ZERO_FILL);
```

An edge detect shows
sharp transitions
between colors.

The Kernel class has some predefined matrices for image sharpening. These matrices are

SHARPEN3x3_1, SHARPEN3x3_2, SHARPEN3x3_3. These matrices are defined as follows:

SHARPEN3x3_1:

-1	-1	-1
-1	9	-1
-1	-1	-1

SHARPEN3x3_2:

1	-2	1
-2	5	-2
1	-2	1

SHARPEN3x3_3:

0	-1	0
-1	5	-1
0	-1	0

You can create a Kernel with one of these predefined matrices like this:

```
Kernel sharpen = new Kernel(Kernel.SHARPEN3x3_1);
```

After you create a Kernel, you use it to create a ConvolveOp:

```
ConvolveOp op = new ConvolveOp(sharpen);
```

Because it relies on surrounding pixels, ConvolveOp must have different source and destination images.

LookupOp The LookupOp class provides a simple table lookup to map one pixel value to another.To create a LookupOp, you need to create a LookupTable, which takes an array or byte orshort values. LookupTable itself is an abstract class. You must create either a ByteLookupTableor a ShortLookupTable. The following code fragment creates a LookupOp that reverses colors(0 becomes 255, 1 becomes 254, and so on):

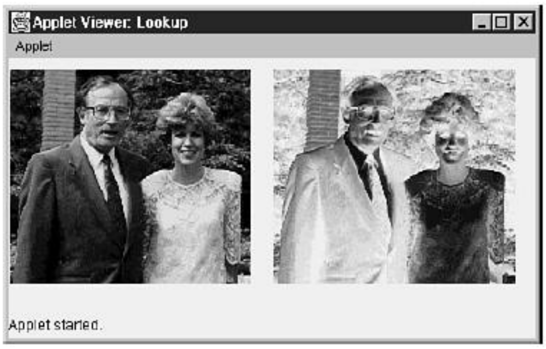
```
short lookupValues[] = new short[256];
for (int i=0; i < lookupValues.length; i++) {
lookupValues[i] =(short) (255 - i);
}
LookupTable table = new ShortLookupTable(0, lookupValues);
LookupOp op = new LookupOp(table);
```

Figure 26.10 shows an image before and after this reverse-color lookup.

LookupOp can have the same source and destination images.

RescaleOp The RescaleOp class enables you to change colors in an image based on a coefficient and an offset. The name might lead you to believe that it changes the size of an image,but it does not (use the AffineTransformOp for that). You create a RescaleOp by supplying ascaling factor and an offset. Each color in the image is multiplied by the scaling factor and thenadded to the offset. The Constructors for RescaleOp are declared as follows:

Fig 6.28



```
public RescaleOp(float factor, float offset)
public RescaleOp(float[] factors, float[] offsets)
```

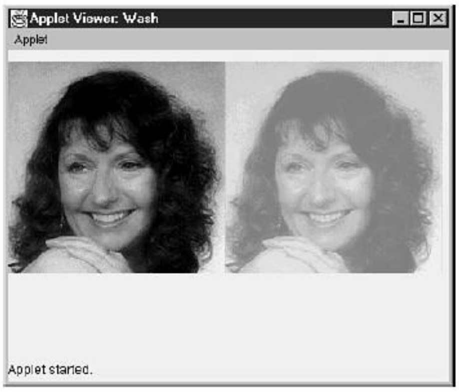
When you pass an array of values to the Constructor, the values apply to each raster channel.

(For an RGB image, that's the red, green, and blue channels.) If you pass singular values, theyapply to all channels. You can perform a "wash" effect by halving the color values and adding 128. The following code fragment creates a wash RescaleOp:

```
RescaleOp op = new RescaleOp(0.5f, 128);
```

Figure 6.29 shows an image before and after the wash effect.

Fig 6.29



Applet started.

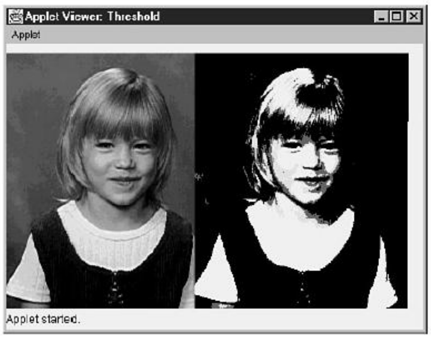
ThresholdOp Rounding out the list of predefined operations, ThresholdOp provides an on-off type filter for colors. You specify a threshold value, a low value, and a high value. Any color less than the threshold value is assigned the low value, and any color higher than the threshold is assigned the high value. The assignments are done on a per-channel basis. (That is, it works on the red, and then the green, and then the blue channels in an RGB image.) You can specify different values for each color channel by providing arrays of threshold, low, and high values.

As you might guess, ThresholdOp really reduces the number of colors in an image. The Constructorsfor ThresholdOp are as follows:

```
        public ThresholdOp(float threshold, float low, float high)
public ThresholdOp(float[] thresholds, float[] lows, float[] highs)
```

Figure 26.12 shows an image with a threshold of 128, a low value of 0, and a high value of 255. You might think that it would produce a completely black and white picture, but there are a few patches of color. These occur when some channels go to 0 and others go to 255.

Fig 6.30



Manipulating Buffered Images:

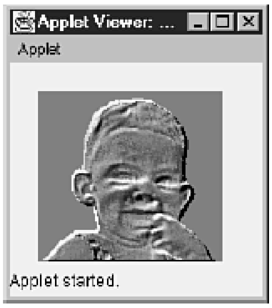
Sometimes we may want to manipulate the pixels in a buffered image without using a filter. We can use getRGB and setRGB to manipulate pixels directly.

Listing 6.22 Source Code for Emboss.java

```
import java.awt.*;
import java.applet.*;
import java.awt.geom.*;
import java.awt.font.*;
import java.awt.image.*;
import java.net.URL;
public class Emboss extends Applet
{
public void paint(Graphics g)
{
// Get the Graphics2D object
Graphics2D newG = (Graphics2D) g;
// Load an image to display
URL imgURL = null;
try {
imgURL = new URL(getDocumentBase(), “katyface.gif”);
} catch (Exception ignore) {
}
Image img = getImage(imgURL);
MediaTracker tracker = new MediaTracker(this);
try {
tracker.addImage(img, 0);
tracker.waitForAll();
} catch (Exception e) {
e.printStackTrace();
}
// Normally you would create a buffered image and set the individual
// pixels yourself, but you can also use an existing image. Rather
than
// trying to convert the loaded image into a BufferedImage, it is
easier
// (from a programming standpoint) to just get a Graphics2D
object for
// the BufferedImage and draw the image into it
int width = img.getWidth(this);
int height = img.getHeight(this);
// Create a buffered version of the image by creating a graphics
// context and drawing into it.
BufferedImage buff = new BufferedImage(width,
height, BufferedImage.TYPE_INT_ARGB);
Graphics tempGr = buff.createGraphics();
tempGr.drawImage(img, 0, 0, this);
// Create a buffered image to hold the resulting embossed image
BufferedImage outBuff = new BufferedImage(width,
height, BufferedImage.TYPE_INT_ARGB);
embossImage(buff, outBuff);
newG.drawImage(outBuff, 100, 100, this);
}
// To emboss an image, you start with a completely gray
destination image.
// For each pixel in the source image, look at pixels to the upper-
left and
// lower-right. Figure out the change in red, green, and blue
between the
// upper-left and lower-right and look at the maximum change
(either maximum
// positive or maximum negative) for any color component. For
example,
// if the green changed by -5, blue changed by 10 and red
changed by
// -100, the maximum change would be -100 (the red, which
changed the most).
//
// Now, add the amount of change to 128 (the gray level) and
create a
// pixel in the destination image with red, green, and blue values
equal
// to the new gray level. Make sure you adjust the gray level so it
can't
// be less than 0 or more than 255.
//
//
public void embossImage(BufferedImage srcImage,
BufferedImage destImage)
{
```

```
int width = srcImage.getWidth();
int height = srcImage.getHeight();
// Loop through every pixel
for (int i=0; i < height; i++) {
for (int j=0; j < width; j++) {
// Assume that the upper-left and lower-right are 0
int upperLeft = 0;
int lowerRight = 0;
// If the pixel isn’t on the upper or left edge, get the upper-left
// pixel (otherwise, the upper-left for edge pixels is the default of 0)
if ((i > 0) && (j > 0)) {
// The & 0xffffff strips off the upper 8 bits, which is the transparency
upperLeft = srcImage.getRGB(j-1, i-1)
& 0xffffff;
}
// If the pixel isn’t on the bottom or right edge, get the lower-right
// pixel (otherwise, the lower-right for egde pixels is the default of 0)
if ((i < height-1) && (j < width-1)) {
// The & 0xffffff strips off the upper 8 bits, which is the transparency
lowerRight = srcImage.getRGB(j+1, i+1)
& 0xffffff;
}
// Get the differences between the red, green and blue pixels
int redDiff = ((lowerRight >> 16) & 255) -
((upperLeft >> 16) & 255);
int greenDiff = ((lowerRight >> 8) & 255) -
((upperLeft >> 8) & 255);
int blueDiff = (lowerRight & 255) -
(upperLeft & 255);
// Figure out which color had the greatest change
int diff = redDiff;
if (Math.abs(greenDiff) > Math.abs(diff))
diff=greenDiff;
if (Math.abs(blueDiff) > Math.abs(diff))
diff=blueDiff;
// Add the greatest change to a medium gray
int greyColor = 128 + diff;
// If the gray is too high or too low, make it fit in the 0-255 range
if (greyColor > 255) greyColor = 255;
if (greyColor < 0) greyColor = 0;
// Create the new color, and don’t forget to add in a transparency
// of 0xff000000 making the image completely opaque
int newColor = 0xff000000 + (greyColor << 16) +
(greyColor << 8) + greyColor;
destImage.setRGB(j, i, newColor);
}
}
}
}
```

Fig 6.31



Applet started.

Transparency:

We can create a color with a transparent component with one of the following Constructors for
Color:

```
public Color(int red, int green, int blue, int alpha)
public Color(int rgba, boolean hasAlpha)
public Color(float red, float green, float blue, float alpha)
```

Listing 6.23 Source Code for AlphaDraw.java

```
import java.awt.*;
import java.applet.*;
import java.awt.geom.*;
import java.awt.image.BufferedImage;
import java.net.URL;
public class AlphaDraw extends Applet
{
public void paint(Graphics g)
```

continues

Transparency

```
{
Graphics2D newG = (Graphics2D) g;
// Create a partially transparent blue
Color transBlue = new Color(0, 0, 255, 128);
newG.setColor(transBlue);
GeneralPath path = new GeneralPath();
path.moveTo(60, 0);
path.lineTo(50, 300);
path.curveTo(160, 230, 270, 140, 400, 100);
newG.fill(path);
Color transRed = new Color(255, 0, 0, 128);
newG.setColor(transRed);
path = new GeneralPath();
path.moveTo(200, 300);
path.lineTo(10, 100);
path.lineTo(300, 40);
path.lineTo(200, 300);
newG.fill(path);
}
}
```

Clipping:

To Clip create a shape and call the clip method in Graphics2D:

```
public clip(Shape shape)
```

Following source code shows how to use a text string as a clipping area. Note that you need to translatethe string before clipping with it. Make sure the johnpat2.gif file is in the same directory asyour applet.

Source Code for TextClip.java

```
import java.awt.*;
import java.applet.*;
import java.awt.geom.*;
import java.awt.font.*;
import java.awt.image.BufferedImage;
import java.net.URL;
public class TextClip extends Applet
{
public void paint(Graphics g)
{
// Get the Graphics2D object
Graphics2D newG = (Graphics2D) g;
// Load an image to use as the texture
URL imgURL = null;
try {
imgURL = new URL(getDocumentBase(), “johnpat2.gif”);
} catch (Exception ignore) {
}
Image img = getImage(imgURL);
MediaTracker tracker = new MediaTracker(this);
try {
```

```
tracker.addImage(img, 0);
tracker.waitForAll();
} catch (Exception e) {
e.printStackTrace();
}
int width = img.getWidth(this);
int height = img.getHeight(this);
Font bigfont = new Font(“Serif”, Font.BOLD, 60);
StyledString ssGrahams = new StyledString(“The Grahams”,
bigfont);
// Set the clipping area to be the text string
AffineTransform transform = new AffineTransform();
transform.translate(0, 100);
Shape clipShape = transform.createTransformedShape(
ssGrahams.getStringOutline());
newG.clip(clipShape);
// Draw the image over the clipped area
newG.drawImage(img, 0, 0, width*2, height*2, this);
}
}
```

6.17 SUMMARY

This capters covers Java Foundation Classes, adding Buttons with JFC ,adding ToolTips and Icons , Pop-Up Menus , Borders , Check Boxes and Radio Buttons , applying CheckBoxPanel to Change Text Alignment, Tabbed Panes, Sliders, Progress Bars, Menus and Toolbars, Lists and Combo Boxes, Using Tables, Trees.

6.18 QUESTIONS

1. How will you add tooltips and icons to diifernt GUI component?
2. Explain different types of borders in java.
3. Explain Menus and toolbars in java.
4. How will you align your text using Checkboxpanel.

*****

7

INTERNATIONALIZATION

Unit Structure

7.1 INTERNATIONALIZATION SCENARIO

Jayant Programmer is a Java developer for Company X. His distributed sales application is a hugesuccess in India, where his company is based, mainly because he follows good objectorienteddesign and implementation: keeping his objects portable, reusable, and independent.One day, Company X decides to start selling its product in Japan. Jayant Programmer, who doesnot know Japanese, gets a Java-literate translator to go through his code and make all the necessarychanges using some custom Japanese language character set that Jayant doesn't reallyunderstand. But he happily compiles this Japanese-language version of his code and sends itoff to Japan, where it is a big success. Encouraged by this result, Company X starts movinginto other markets; France and Canada are next. To Jayant's dismay, he finds that he has to maintainseveral completely different versions of his code because France and Canada, althoughthey share a common language, have a completely different culture! Poor Jayant now has fivecompiled versions of his code: an American English, Japanese, French, Canadian French, andCanadian English. Now, when he makes even the slightest change to his code, he has to makethe same change five times, and then hire several translators to make language changes directlyin the source code. Clearly, Jayant is in an unacceptable situation.

7.2 WHAT IS INTERNATIONALIZATION?

In the previous scenario, Jayant Programmer is said to have written a *myopic* program, one that is only suited to one locale. A locale is a region (usually geographic, but not necessarily so) thatshares customs, culture, and language.

Each of the five versions of Jayant’s program was localizedfor one specific locale and was unusable outside that locale without major alteration. This violatesthe fundamental principle of OOP design, because Jayant’s program is no longer portable orreusable. The process of isolating the culture-dependent code (text, pictures, and so on) fromthe language-independent code (the actual functionality of the program) is called internationalization.After a program has been through this process, it can easily be adapted to any localewith a minimum amount of effort. Version 1.1 of the Java language added built-in support forinternationalization, which makes writing truly portable code easy.

7.3 JAVA SUPPORT FOR INTERNATIONALIZATION

Internationalization required several changes to the Java language. In the past, writing internationalizedcode required extra effort and was substantially more difficult than writing myopic code. One of the design goals was to reverse this paradigm. Java seeks to make writing internationalized code easier than its locale-specific counterpart. Internationalization mainly affects three packages:

-java.util Includes the Locale class. A Locale encapsulates certain information abouta locale, but does not provide the actual locale-specific operations. Rather, affectedmethods can now be passed a Locale object as a parameter that will alter their behavior. If no Locale is specified, a default Locale is taken from the environment. This packagealso provides support for ResourceBundles, objects that encapsulate locale-sensitive datain a portable, independent way.

-java.io All of the classes in java.io that worked with InputStreams and OutputStreamshave corresponding classes that work with class Reader and Writer. Readers andWriters work like Streams, except they are designed to handle 16-bit Unicode charactersinstead of 8-bit bytes.

-java.text An entirely new package that provides support for manipulating variouskinds of text. This includes collating (sorting) text, formatting dates and numbers, andparsing language-sensitive data.

The Locale class:

A Locale object encapsulates information about a specific locale. This consists of just enoughinformation to uniquely identify the locale’s region. When a locale-sensitive method is passed aLocale object as a parameter, it attempts to modify its behavior for that particular locale. ALocale is initialized with a language code, a country code, and an optional variant code.

Thesethree things define a region, although you need not specify all three. For example, you couldhave a Locale object for American English, California variant. If you ask the Calendar classwhat the first month of the year is, the Calendar tries to find a name suitable for CalifornianAmerican English. Because month names are not affected by what state you are in, theCalendar class has no built-in support for Californian English, and it tries to find a best fit.It next tries American English, but because month names are constant in all English-speakingcountries, this fails as well. Finally, the Calendar class returns the month name thatcorresponds to the English Locale. This best-fit lookup procedure allows the programmercomplete control over the granularity of internationalized code.You create a Locale object using the following syntax:

```
Locale theLocale = new Locale(“en”, “US”);
```

where “en” specifies English, and “US” specifies United States. These two-letter codes are used internally by Java programs to identify languages and countries. They are defined by the ISO- 639 and ISO-3166 standards documents respectively.

Currently, the JDK supports the following language and country combinations in all of its locale-sensitive classes, such as Calendar, NumberFormat, and so on. This list may change in the future, so be sure to check the latest documentation (see Table.7.1).
Java Support for Internationalization

Table.7.1 Locales Supported by the JDK

Locale Country	**Language**
da_DK	Denmark Danish
DE_AT	Austria German
de_CH	Switzerland German
de_DE	Germany German
el_GR	Greece Greek
en_CA	Canada English
en_GB	United Kingdom English
en_IE	Ireland English
en_US	United States English
es_ES	Spain Spanish
fi_FI	Finland Finnish
fr_BE	Belgium French
fr_CA	Canada French
fr_CH	Switzerland French
fr_FR	France French

it_CH	Switzerland Italian
it_IT	Italy Italian
ja_JP	Japan Japanese
ko_KR	Korea Korean
nl_BE	Belgium Dutch
nl_NL	Netherlands Dutch
no_NO	Norway Norwegian (Nynorsk)
no_NO_B	Norway Norwegian (Bokmål)
pt_PT	Portugal Portuguese
sv_SE	Sweden Swedish
tr_TR	Turkey Turkish
zh_CN	China Chinese(Simplified)
zh_TW	Taiwan Chinese (Traditional)

667

Programmers can also create their own custom Locales, simply by specifying a unique sequence of country, language, variant. Multiple variants can be separated by an underscorecharacter. To create a variant of Californian American English running on a Windows machine,use the following code:

```
Locale theLocale = new Locale(“en”, “US”, “CA_WIN”);
```

Remember that methods that do not understand this particular variant will try to find a best fitmatch, in this case probably “en_US”.The two-letter abbreviations listed here are not meant to be displayed to the user; they aremeant only for internal representation. For display, use one of the Locale methods listed in Table.7.2. You will notice that these methods are generally overloaded so that you can get theparameter either for the current locale or the one specified.

Table.7.2 Locale Display Methods

Method Name	**Description**

getDisplayCountry()
getDisplayCountry(Locale)
	Country name, localized for default Locale, or specifiedLocale.

getDisplayLanguage()
getDisplayLanguage(Locale)
	Language name, localized for default Locale, orspecified Locale.

getDisplayName()
getDisplayName(Locale)
	Name of the entire locale, localized for default Locale,or specified Locale.

getDisplayVariant()	

getDisplayVariant(Locale)	Name of the Locale’s variant. If the localized name isnot found, this returns the variant code.

These methods are very useful when you want to have a user interact with a Locale object.

Here’s an example of using the getDisplayLanguage() method:

```
Locale.setDefault( new Locale(“en”, “US”) ); //Set default Locale to American
//English
Locale japanLocale = new Locale(“ja:, “JP”); //Create locale for Japan
System.out.println( japanLocale.getDisplayLanguage() );
System.out.println( japanLocale.getDisplayLanguage( Locale.FRENCH ) );
```

This code fragment prints out the name of the language used by japanLocale. In the first case, it is localized for the default Locale, which has been conveniently set to American English. The output would therefore be Japanese. The second print statement localizes the language name for display in French, which yields the output Japonais. All of the Locale “display” methods Java Support for Internationalizationuse this same pattern. Almost all Internationalization API methods allow you to explicitlycontrol the Locale used for localization, but in most cases, you’ll just want to use the defaultLocale.Another thing to note in the preceding example is the use of the static constant Locale.FRENCH.The Locale class provides a number of these useful constants, each of which is a shortcut forthe corresponding Locale object. A list of these objects is shown in Table.7.3.

Table.7.3 Locale Static Objects

Constant Name	**Locale Shortcut for**
CANADA	English Canadanew Locale(“en”, “CA”, “”)
CANADA_FRENCH	French Canadanew Locale(“fr”, “CA”, “”)
CHINA SCHINESE PRC	Chinese (Simplified) new Locale(“zh”, “CN”, “”)
CHINESE	Chinese Language new Locale(“zh”, “”, “”)
ENGLISH	English Languagenew Locale(“en”, “”, “”)
FRANCE	France new Locale(“fr”, “FR”, “”)

FRENCH	French Language new Locale(“fr”, “”, “”)
GERMAN	German Languagenew Locale(“de”, “”, “”)
GERMANY	Germanynew Locale(“de”, “DE”, “”)
ITALIAN	Italian Language new Locale(“it”, “”, “”)
ITALY	Italy new Locale(“it”, “IT”, “”)
JAPAN	Japannew Locale(“jp”, “JP”, “”)
JAPANESE	Japanese Language new Locale(“jp”, “”, “”)
KOREA	Korea new Locale(“ko”, “KR”, “”)
KOREAN	Korean Language new Locale(“ko”, “”, “”)
TAIWAN TCHINESE (Traditional Chinese)	Taiwan new Locale(“zh”, “TW”, “”)
UK	Great Britain new Locale(“en”, “GB”, “”)
US	United States new Locale(“en”, “US”, “”)

7.4INPUT-OUTPUT (I/O) FOR INTERNATIONALIZATION

Originally, the java.io package operated exclusively on byte streams, a continuous series of8-bit quantities. However, Java’s Unicode characters are 16 bits, which makes using them withbyte streams difficult. Therefore, if you look at the java.io package, there is also a wholeseries of 16-bit character stream Readers and Writers, which correspond to the oldInputStream and OutputStream. The two sets of classes can work together or separately, dependingon whether your program needs to input or output text of any kind.

Character Set Converters:

The way in which characters are represented as binary numbers is called an encoding scheme. The most common scheme used for English text is called the ISO Latin-1 encoding. The set of characters supported by any one encoding is said to be its character set, which includes all possible characters that can be represented by the encoding. Usually, the first 127 codes of an encoding correspond to the almost universally accepted ASCII character set, which includes all of the standard characters and punctuation marks. Nevertheless, most encodings can varyradically, especially because some, like Chinese and Japanese encodings, have character sets that bear little resemblance to English!

Luckily, Java 1.1 provides classes for dealing with all of the most common encodings around.The ByteToCharConverter and CharToByteConverter classes are responsible for performingvery complex conversions to and from the standard Unicode characters supported by Java.Each encoding scheme is given its own label by which it can be identified. A complete list ofsupported encodings and their labels is shown in Table.7.4.

Table.7.4 JDK 1.1–Supported Character Encodings

Label	Encoding Scheme Description
8859_1	ISO Latin-1
8859_2	ISO Latin-2
8859_3	ISO Latin-3
8859_4	ISO Latin-4
8859_5	ISO Latin/Cyrillic
8859_6	ISO Latin/Arabic
8859_7	ISO Latin/Greek
8859_8	ISO Latin/Hebrew
8859_9	ISO Latin-5
Big5	Big 5 Traditional Chinese
CNS11643	CNS 11643 Traditional Chinese
Cp1250	Windows Eastern Europe/Latin-2
Cp1251	Windows Cyrillic
Cp1252	Windows Western Europe/Latin-1
Cp1253	Windows Greek
Cp1254	Windows Turkish
Cp1255	Windows Hebrew
Cp1256	Windows Arabic
Cp1257	Windows Baltic
Cp1258	Windows Vietnamese
Cp437	PC Original
Cp737	PC Greek
Cp775	PC Baltic
Cp850	PC Latin-1
Cp852	PC Latin-2
Cp855	PC Cyrillic
Cp857	PC Turkish
Cp860	PC Portuguese
Cp861	PC Icelandic
Cp862	PC Hebrew
Cp863	PC Canadian French
Cp864	PC Arabic
Cp865	PC Nordic

Cp866	PC Russian
Cp869	PC Modern Greek
Cp874	Windows Thai
EUCJIS	apanese EUC
GB2312	GB2312-80 Simplified Chinese
JIS	JIS
KSC5601	KSC5601 Korean
MacArabic	Macintosh Arabic
MacCentral	Europe Macintosh Latin-2
MacCroatian	Macintosh Croatian
MacCyrillic	Macintosh Cyrillic
MacDingbat	Macintosh Dingbat
MacGreek	Macintosh Greek
MacHebrew	Macintosh Hebrew
MacIceland	Macintosh Iceland
MacRoman	Macintosh Roman
MacRomania	Macintosh Romania
MacSymbol	Macintosh Symbol
MacThai	Macintosh Thai
MacTurkish	Macintosh Turkish
MacUkraine	Macintosh Ukraine
SJIS	PC and Windows Japanese
UTF8	Standard UTF-8

Java also provides ways for developers to create their own encodings and to create converters for already-existing but unsupported encodings. The details of how character conversion isdone are actually quite complex, and those who are interested are referred to Java’s Webpages.

Readers and Writers :

Character streams make heavy use of character set converters. Fortunately, they also hide theunderlying complexity of the conversion process, making it easy for Java programs to be writtenwithout knowledge of the internationalizing process. Again, you see that programs areinternationalized by default.The advantages of using character streams over byte streams are many. Although they havethe added overhead of doing character conversion on top of byte reading, they also allow formore efficient buffering. Byte streams are designed to read information one byte at a time, Table.7.4 Continued

Label Encoding Scheme Description:

while character streams read one buffer at a time. According to Sun, this, combined with a new efficient locking scheme, more than compensates for the speed loss caused by the conversion process. Every input or output stream in the old class hierarchy now has a corresponding Reader or Writer class that performs similar functions using character streams table 5).

Input/Output Streams and Corresponding Reader and WriterClasses (from Sun Microsystems, Inc.)

Table 7.5.

Byte Stream Class (InputStream/ OutputStream)	**Corresponding** Character Stream Class (Reader/Writer)	**Function**
InputStream	Reader	Abstract class from which all other classes inherit methods, and so on
BufferedInputStream	BufferedReader	Provides a buffer for input operations
LineNumberInputStream	LineNumberReader	Keeps track of line numbers
ByteArrayInputStream	CharArrayReader	Reads from an array
N/A	InputStreamReader	Translates a byte stream into a character stream
FileInputStream	FileReader	Allows input from a file on disk
FilterInputStream	FilterReader	Abstract class for filtered input
PushbackInputStream	PushbackReader	Allows characters to be pushed back into the stream
PipedInputStream	PipedReader	Reads from a process pipe
StringBufferInputStream	StringReade	Reads from a String
OutputStream	Writer	Abstract class for characteroutput streams
BufferedOutputStream	BufferedWriter	Buffers output, uses platform’s line separator
ByteArrayOutputStream	CharArrayWriter	Writes to a character array
FilterOutputStream	FilterWriter	Abstract class for filtered character output
N/A	OutputStreamWriter	Translates a character stream into a byte stream
FileOutputStream	FileWriter	Translates a character stream into a byte file
PrintStream	PrintWriter	Prints values and objects to a Writer
PipedOutputStream	PipedWriter	Writes to a PipedReader
N/A	StringWriter	Writes to a String

7.5 THE NEW PACKAGE JAVA.TEXT

The most advanced and complex Internationalization API features are found in the java.text package. They include many classes for formatting and organizing text in a languageindependent way Text collating, on the other hand, is the process of sorting text according to particular rules. InEnglish, sorting in alphabetical order is relatively easy because English lacks many special characters (such as accents) that could complicate things. In French, however, things are not so simple. Two words that look very similar (like péché and pêche) have entirely different .Byte Stream Class Corresponding Function (InputStream/ Character Stream OutputStream) Class (Reader/Writer)meanings. Which should come first alphabetically? And what about characters like hyphenationor punctuation? The Java Collation class provides a way of defining language-specificsort criteria in a robust, consistent manner.Text boundaries can also be ambiguous across languages. Where do words, sentences, andparagraphs begin and end? In English, a period generally marks the end of a sentence, but isthis always the case? Certainly not. The TextBoundary and CharacterIterator classes canintelligently break up text into various sub-units based on language-specific criteria. Java comeswith built-in support for some languages, but you can always define your own set of rules, aswell. TextBoundary works by returning the integer index of boundaries that occur within aString, as demonstrated by the following example, which breaks up a String by words:

```
String str = “This is a line of text. It contains many words, sentences, and formatting.”;

TextBoundary byWord = TextBoundary.getWordBreak();
int from, to;
from = byWord.first();
while( (to = byWord.next()) != DONE ) {
System.out.println( byWord.getText().substring(from, to) );
from = to;
}
```

7.6 AN EXAMPLE: INTERNATIONALTEST

The application is a very simple one. It takes up to three command-line parameters that specify a locale. It uses this information to

1. Display some information about the default locale and the one entered

2. Try to load a ResourceBundle corresponding to the specified locale and print out whatthe Bundle contains

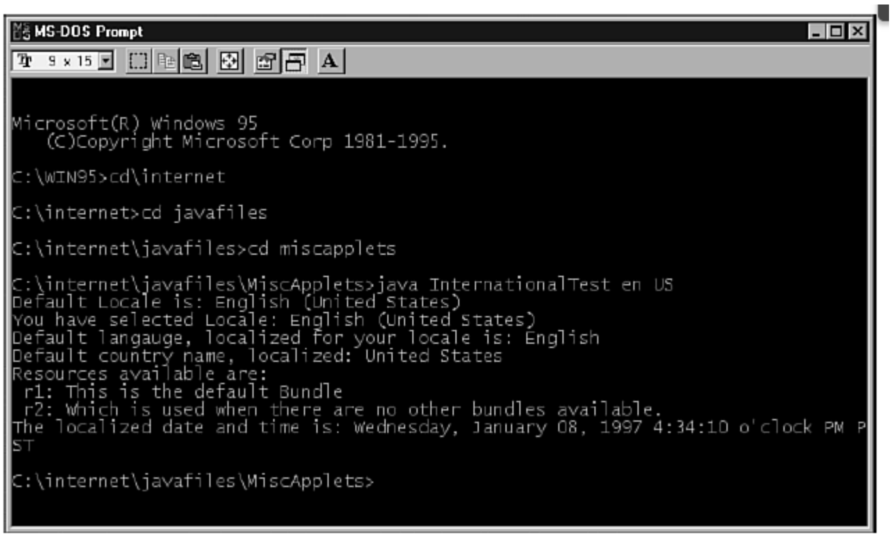
3. Display the date, localized to the specified locale

Listing 7.1 InternationalTest.java

```
import java.util.*;
import java.lang.*;
import java.text.DateFormat;
class InternationalTest extends Object {
public static void main(String args[]) {
String lang = “”, country = “”, var = “”;
try {
lang = args[0];
country = args[1];
var = args[2];
} catch(ArrayIndexOutOfBoundsException e) {
if( lang.equals(“”) ) {
System.out.println(“You must specify at least one parameter”);
System.exit(1);
}
}
Locale locale = new Locale(lang, country, var);
Locale def = Locale.getDefault();
System.out.println( “Default Locale is: “+ def.getDisplayName()
);
System.out.println(“You have selected Locale:
“+locale.getDisplayName() );
System.out.println(“Default language, localized for your locale is:
“ +
def.getDisplayLanguage( locale ) );
System.out.println(“Default country name, localized: “ +
locale ) );
ClassLoader loader = null;
ResourceBundle bundle = null;
try {
bundle = ResourceBundle.getResourceBundle( “TestBundle”,
locale, loader );
} catch( MissingResourceException e) {
System.out.println( “No resources available for that locale.” );
} finally {
System.out.println( “Resources available are: “);
System.out.println(“ r1: “ + bundle.getString(“r1”) );
System.out.println(“ r2:” + bundle.getString(“r2”) );
}
DateFormat myFormat =
DateFormat.getDateTimeFormat(DateFormat.FULL,
DateFormat.FULL, locale);
Calendar myCalendar = Calendar.getDefault( locale );
System.out.println("The localized date and time is: " +
myFormat.format( myCalendar.getTime() ) );
}
}
```

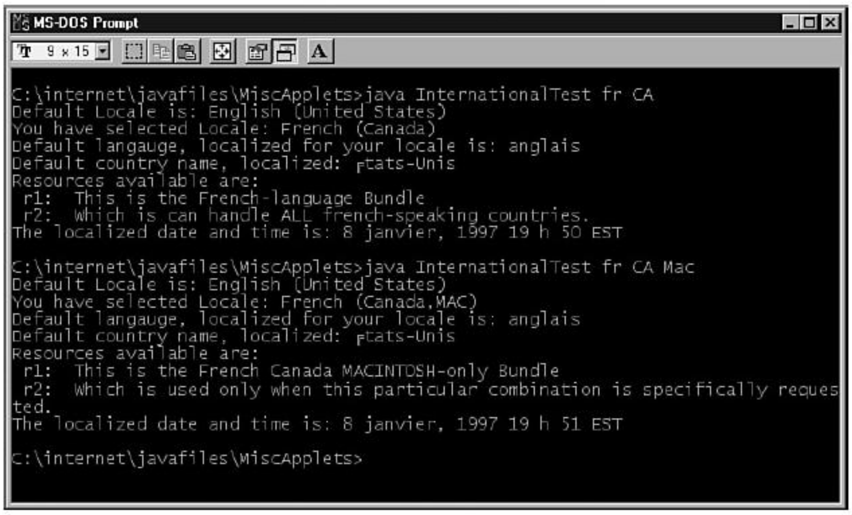
output from the InternationalTest program.
American English locale.

Fig 7.1 Canadian French and Canadian French Macintosh locales.



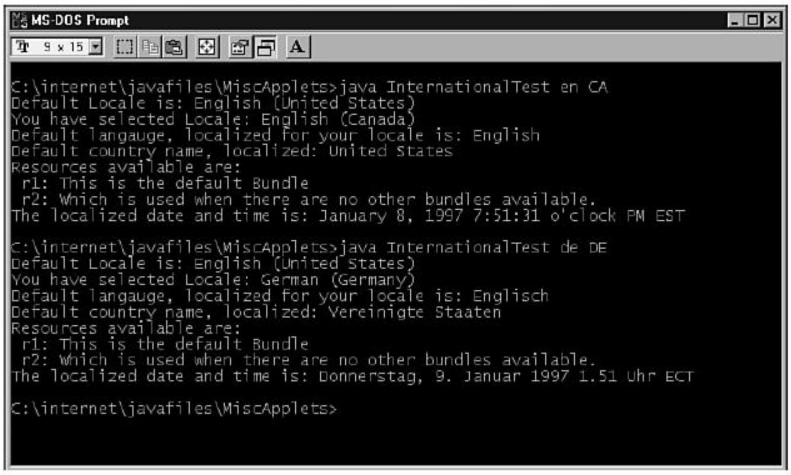
```
Microsoft(R) Windows 95
   (C)Copyright Microsoft Corp 1981-1995.

C:\WIN95>cd\internet

C:\internet>cd javafiles

C:\internet\javafiles>cd miscapplets

C:\internet\javafiles\MiscApplets>java InternationalTest en US
Default Locale is: English (United States)
You have selected Locale: English (United States)
Default langauge, localized for your locale is: English
Default country name, localized: United States
Resources available are:
 r1: This is the default Bundle
 r2: Which is used when there are no other bundles available.
The localized date and time is: Wednesday, January 08, 1997 4:34:10 o'clock PM P
ST

C:\internet\javafiles\MiscApplets>
```

Fig 7.2



```
C:\internet\javafiles\MiscApplets>java InternationalTest fr CA
Default Locale is: English (United States)
You have selected Locale: French (Canada)
Default langauge, localized for your locale is: anglais
Default country name, localized: ╔tats-Unis
Resources available are:
 r1:  This is the French-language Bundle
 r2:  Which is can handle ALL french-speaking countries.
The localized date and time is: 8 janvier, 1997 19 h 50 EST

C:\internet\javafiles\MiscApplets>java InternationalTest fr CA Mac
Default Locale is: English (United States)
You have selected Locale: French (Canada,MAC)
Default langauge, localized for your locale is: anglais
Default country name, localized: ╔tats-Unis
Resources available are:
 r1:  This is the French Canada MACINTOSH-only Bundle
 r2:  Which is used only when this particular combination is specifically reques
ted.
The localized date and time is: 8 janvier, 1997 19 h 51 EST

C:\internet\javafiles\MiscApplets>
```

Fig 7.3
Canadian English andGermany locales.



```
C:\internet\javafiles\MiscApplets>java InternationalTest en CA
Default Locale is: English (United States)
You have selected Locale: English (Canada)
Default langauge, localized for your locale is: English
Default country name, localized: United States
Resources available are:
 r1: This is the default Bundle
 r2: Which is used when there are no other bundles available.
The localized date and time is: January 8, 1997 7:51:31 o'clock PM EST

C:\internet\javafiles\MiscApplets>java InternationalTest de DE
Default Locale is: English (United States)
You have selected Locale: German (Germany)
Default langauge, localized for your locale is: Englisch
Default country name, localized: Vereinigte Staaten
Resources available are:
 r1: This is the default Bundle
 r2: Which is used when there are no other bundles available.
The localized date and time is: Donnerstag, 9. Januar 1997 1.51 Uhr ECT

C:\internet\javafiles\MiscApplets>
```

7.7 SUMMARY

This capters covers internationalization Scenario, what Is Internationalization, Java Support for Internationalization, Input-Output (I/O) for Internationalization and new Package java.text.

7.8 QUESTIONS

5. What is Internationalization?
6. Explain Locals.
7. Explain Readers and Writers.
8. Write a source code to create International test.

*****

8

COMMUNICATION AND NETWORKING, TCP SOCKETS, UDP SOCKETS, JAVA.NET, JAVA SECURITY

Unit Structure

8.1 OVERVIEW OF TCP/IP

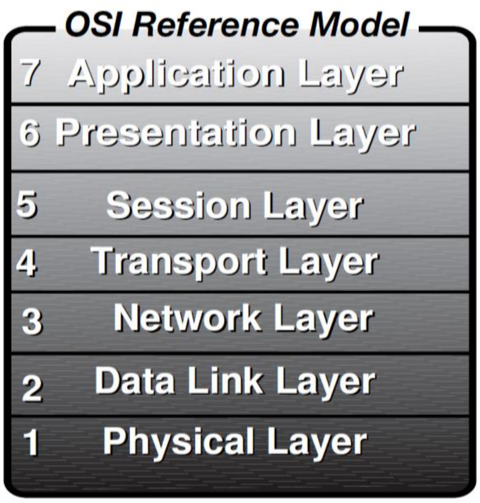
TCP/IP is a suite of protocols that interconnects the various systems on the Internet. TCP/IPprovides a common programming interface for diverse and foreign hardware. The suite supportsthe joining of separate physical networks implementing different network media. TCP/IPmakes a diverse, chaotic, global network like the Internet possible.

OSI Reference Model:

The network protocol architecture known as the *Open Systems Interconnect (OSI) ReferenceModel* is often used to describe network systems. The OSI scheme was one part of a largerproject by the International Organization for Standardization (ISO). The OSI protocols neverproved as successful as TCP/IP, making the Reference Model perhaps the most enduringaspect of this ISO endeavor.The model consists of seven layers providing specific functionality. Each layer has definedcharacteristics, and together the whole enables network communication. The software implementationof such a layered model is appropriately termed a *protocol stack*.The OSI model is illustrated in Figure 30.1. User applications insert information into one layerand each encapsulates the data until the last is reached. The information is then transmitted tothe destination, sometimes having the layers translated from the bottom up as the data is transported.

The following layers have specific roles, each refraining from intruding into the domain of theother, all depending upon the others:

Fig 8.1



-Application Layer Contains network applications within which people interact, such asmail, file transfer, and remote login.

-Presentation Layer Creates common data structures.

-Session Layer Manages connections between network applications.

-Transport Layer Ensures that data is received exactly as it is sent.

-Network Layer Routes data through various physical networks while traveling to aknown host.

-Data Link Layer Transmits and receives packets of information reliably across auniform physical network.

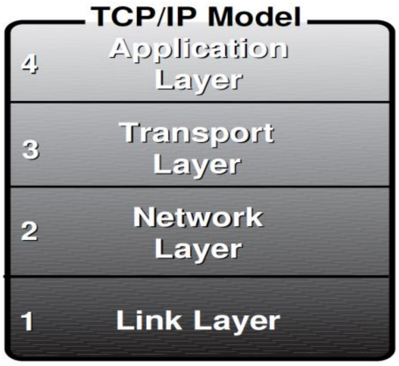
-Physical Layer Defines the physical properties of the network, such as voltage levels,cable types, and interface pins.

TCP/IP Network Model:

The OSI model helps when trying to understand the TCP/IP communication architecture. When viewed as a layered model, TCP/IP is usually seen as being composed of four layers:

Application
Network
Transport
Link

Fig 8.2



As in the OSI model, each TCP/IP layer plays a specific role, each of which is described in the following four sections.

Application Layer Network applications depend on the definition of a clear dialog. In a clientserversystem, the client application knows how to request services, and the server knows howto appropriately respond. Protocols that implement this layer include HTTP, FTP, and Telnet.

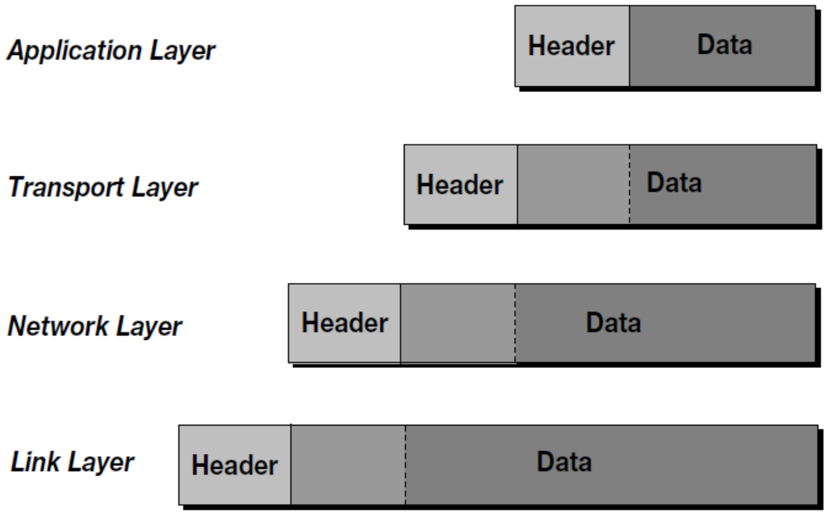
Transport Layer The Transport Layer enables network applications to obtain messages over clearly defined channels and with specific characteristics. The two protocols within the TCP/IP suite that generally implement this layer are Transmission Control Protocol (TCP) and User Datagram Protocol (UDP).

Network Layer The Network Layer enables information to be transmitted to any machine on the contiguous TCP/IP network, regardless of the different physical networks that intervene.

Internet Protocol (IP) is the mechanism for transmitting data within this layer.

Link Layer The Link Layer consists of the low-level protocols used to transmit data to machines on the same physical network. Protocols that aren't part of the TCP/IP suite, such asEthernet, Token Ring, FDDI, and ATM, implement this layer.

Data within these layers is usually encapsulated with a common mechanism: protocols have a header that identifies meta information such as the source, destination, and other attributes, and a data portion that contains the actual information. The protocols from the upper layers are encapsulated within the data portion of the lower ones. When traveling back up the protocol stack, the information is reconstructed as it is delivered to each layer. Figure 8.3 shows this concept of encapsulation.



8.2 TCP/IP PROTOCOLS

Three protocols are most commonly used within the TCP/IP scheme, and a closer investigation of their properties is warranted. Understanding how these three protocols (IP, TCP, andUDP) interact is critical to developing network applications.

Internet Protocol (IP) :

IP is the keystone of the TCP/IP suite. All data on the Internet flows through IP packets, thebasic unit of IP transmissions. IP is termed a connectionless, unreliable protocol. As aconnectionless protocol, IP does not exchange control information before transmitting data to aremote system—packets are merely sent to the destination with the expectation that they willbe treated properly. IP is unreliable because it does not retransmit lost packets or detect corrupteddata. These tasks must be implemented by higher level protocols, such as TCP.IP defines a universal addressing scheme called IP addresses. An IP address is a 32-bit number,and each standard address is unique on the Internet. Given an IP packet, the informationcan be routed to the destination based upon the IP address defined in the packet header. IPaddresses are generally written as four numbers, between 0 and 255, separated by a period (forexample, 124.148.157.6).

While a 32-bit number is an appropriate way to address systems for computers, humans understandablyhave difficulty remembering them. Thus, a system called the Domain Name System(DNS) was developed to map IP addresses to more intuitive identifiers and vice versa. You canuse www.netspace.org instead of 128.148.157.6.

It is important to realize that these domain names are not used or understood by IP. When an application wants to transmit data to another machine on the Internet, it must first translate the domain name to an IP address using the DNS. A receiving application can perform a reversetranslation, using the DNS to return a domain name given an IP address. There is not a one-to one

correspondence between IP addresses and domain names: a domain name can map tomultiple IP addresses, and multiple IP addresses can map to the same domain name.

Transmission Control Protocol (TCP):

Most Internet applications use TCP to implement the transport layer. TCP provides a reliable, connection-oriented, continuous-stream protocol. The implications of these characteristics are:

- Reliable When TCP segments, the smallest unit of TCP transmissions, are lost orcorrupted, the TCP implementation will detect this and retransmit necessary segments.
- Connection-oriented TCP sets up a connection with a remote system by transmittingcontrol information, often known as a handshake, before beginning a communication. Atthe end of the connect, a similar closing handshake ends the transmission.

- Continuous-stream TCP provides a communications medium that allows for anarbitrary number of bytes to be sent and received smoothly; once a connection has beenestablished, TCP segments provide the application layer the appearance of a continuousflow of data.

Because of these characteristics, it is easy to see why TCP would be used by most Internetapplications. TCP makes it very easy to create a network application, freeing you from worryinghow the data is broken up or about coding error correction routines. However, TCP requiresa significant amount of overhead and perhaps you might want to code routines thatmore efficiently provide reliable transmissions, given the parameters of your application. Furthermore,retransmission of lost data may be inappropriate for your application, because suchinformation’s usefulness may have expired. In these instances, UDP serves as an alternative,described in the following section, “User Datagram Protocol (UDP).”An important addressing scheme that TCP defines is the port. Ports separate various TCPcommunications streams that are running concurrently on the same system. For server applications,which wait for TCP clients to initiate contact, a specific port can be established fromwhere communications will originate. These concepts come together in a programming abstractionknown as sockets.

User Datagram Protocol (UDP):

UDP is a low-overhead alternative to TCP for host-to-host communications. In contrast to TCP,

UDP has the following features:

- Unreliable UDP has no mechanism for detecting errors, nor retransmitting lost orcorrupted information.
- Connectionless UDP does not negotiate a connection before transmitting data.Information is sent with the assumption that the recipient will be listening.
- Message-oriented UDP enables applications to send self-contained messages withinUDP datagrams, the unit of UDP transmission. The application must package allinformation within individual datagrams.

For some applications, UDP is more appropriate than TCP. For instance, with the Network Time Protocol (NTP), lost data indicating the current time would be invalid by the time it was retransmitted. In a LAN environment, Network File System (NFS) can more efficiently provide reliability at the application layer and thus uses UDP.

As with TCP, UDP provides the addressing scheme of ports, allowing for many applications to simultaneously send and receive datagrams. UDP ports are distinct from TCP ports.For example, one application can respond to UDP port 512 while another unrelated service handles TCP port 512.

8.3 UNIFORM RESOURCE LOCATOR (URL)

While IP addresses uniquely identify systems on the Internet, and ports identify TCP or UDP services on a system, URLs provide a universal identification scheme at the application level.

Anyone who has used a Web browser is familiar with URLs, though their complete syntax may not be self-evident. URLs were developed to create a common format of identifying resources on the Web, but they were designed to be general enough so as to encompass applications that predated the Web by decades. Similarly, the URL syntax is flexible enough to accommodate future protocols.

URL Syntax:

The primary classification of URLs is the scheme, which usually corresponds to an application protocol. Schemes include HTTP, FTP, Telnet, and Gopher. The rest of the URL syntax is in a format that depends on the scheme. These two portions of information are separated by a colon:

scheme-name:scheme-info

Thus, while mailto:dwb@netspace.org indicates “send mail to user ‘dwb’ at the machine netspace.org,” ftp://dwb@netspace.org/ means “open an FTP connection to netspace.org and log in as user dwb.”

General URL Format:

Most URLs conform to a general format that follows this pattern:

scheme-name://host:port/file-info#internal-reference

Scheme-name is an URL scheme such as HTTP, FTP, or Gopher. Host is the domain name or IP address of the remote system. Port is the port number on which the service is listening; because most application protocols define a standard port, unless a non-standard port is being used, the port and the colon that delimits it from the host are omitted. File-info is the resource requested on the remote system, which often is a file. However, the file portion may actually execute a server program and it usually includes a path to a specific file on the system.

The internal-reference is usually the identifier of a named anchor within an HTML page. A named anchor enables a link to target a particular location within an HTML page. Usually this is not used, and this token with the # character that delimits it is omitted.

Realize that this general format is very much an over-simplification that only agrees with common use. For more complete information on URLs, read the following resource:
http://www.netspace.org/users/dwb/url-guide.html
Java and URLs

Java provides a very powerful and elegant mechanism for creating network client applications, allowing you to use relatively few statements to obtain resources from the Internet. The java.net package contains the sources of this power, the URL and URLConnection classes.

The Security Manager of Java browsers generally prohibits applets from opening a network connection to a machine other than the one from which the applet was downloaded. This security feature significantly limits what applets can accomplish. This holds true for all Java networking described in this and subsequent chapters. Java applications, however, are under no such restrictions.

8.4 JAVA AND URLS

The URL Class:

This class enables you to easily create a data structure containing all the necessary information to obtain the remote resource. After an URL object has been created, you can obtain the various portions of the URL according to the general format. The URL object also enables you to obtain the remote data.

The URL class has four constructors:

```
public URL(String spec) throws MalformedURLException;
public URL(String protocol, String host, String file)
throws MalformedURLException;
public URL(String protocol, String host, int port, String file)
throws MalformedURLException;
public URL(URL context, String spec)
throws MalformedURLException;
```

The first constructor is the most commonly used and enables you to create an URL object with a simple declaration like:

```
URL myURL = new URL(“http://www.yahoo.com/”);
```

The second and third constructors enable you to specify explicitly the various portions of the URL. The last constructor enables you to use relative URLs. A relative URL only contains part of the URL syntax; the rest of the data is completed from the URL to which the resource is relative. This will often be seen in HTML pages, where a reference to merely more.html means “get more.html from the same machine and directory where the current document resides.”

Here are examples of these constructors:

```
URL firstURLObject - new URL(“http://www.yahoo.com/”);
URL secondURLObject = new URL(“http”,”www.yahoo.com”,”/”);
URL thirdURLObject = new
URL(“http”,”www.yahoo.com”,80,”/”);
URL fourthURLObject = new
URL(firstURLObject,”text/suggest.html”);
```

The first three statements create URL objects that all refer to the Yahoo! home page, while the fourth creates a reference to “text/suggest.html” relative to Yahoo’s home page (such ashttp://www.yahoo.com/text/suggest.html). All of these constructors throw aMalformedURLException, which you will generally want to catch. The example shown later in Listing 8.1 illustrates this. Note that once you create an URL object, you can change to whichresource it points. To accomplish this, you must create a new URL object.

Connecting to an URL:

Now that you’ve created an URL object, you will want to actually obtain some useful data. There are two main avenues of so doing: reading directly from the URL object or obtaining anURLConnection instance from it.Reading directly from the URL object requires less code, but is much less flexible, and it onlyallows a read-only connection. This is limiting, as many Web services enable you to write informationthat will be handled by a server application. The URL class has an openStream() methodthat returns an InputStream object through which the remote resource can be read byte-bybyte.Handling data as individual bytes is cumbersome, so you will often want to embed the returnedInputStream within a DataInputStream object, allowing you to read the input line-by-line. Thiscoding strategy is often referred to as using a decorator, as the DataInputStream decorates theInputStream by providing a more specialized interface. The fo=towing code fragment obtainsan InputStream directly from the URL object and then decorates that stream:

```
        URL whiteHouse = new
URL(“http://www.whitehouse.gov/”);
InputStream undecoratedInput = whiteHouse.openStream();
DataInputStreamdecoratedInput =new
DataInputStream(undecoratedInput);
```

Another more flexible way of connecting to the remote resource is by using theopenConnection() method of the URL class. This method returns an URLConnection object that provides a number of very powerful methods that you can use to customize your connection to the remote resource.For example, unlike the URL class, an URLConnection enables you to obtain both anInputStream and an OutputStream. This has a significant impact upon the HTTP protocol,whose access methods include both GET and POST. With the GET method, an application merelyrequests a resource and then reads the response. The POST method is often used to provideinput to server applications by requesting a resource, writing data to the server with the HTTPrequest body, and then reading the response. In order to use the POST method, you can write toan OutputStream obtained from the URLConnection prior to reading from the InputStream. Ifyou read first, the GET method will be used and a subsequent write attempt will be invalid.The following code fragment demonstrates using an URLConnection object to contact a remoteserver application using the HTTP POST method by writing to an OutputStream decorated by aPrintStream instance. www.javasoft.com makes a CGI server application available to test outthese methods. The code connects to a CGI application which reverses the POST data and thenreads the reversed data from a decorated InputStream.

```
URL reverseURL =new URL(“http://www.javasoft.com/cgi-bin/backwards”);
URLConnection reverseConn = reverseURL.openConnection();
PrintStream output =new PrintStream(reverseConn.getOutputStream());
DataInputStream input =new DataInputStream(reverseConn.getInputStream());
output.println(“string=TexttoReverse”);
String reversedText = input.readLine();
```

8.5 TCP SOCKETS

Sockets are a programming abstraction that isolates your code from the low-levelimplementations of the TCP/IP protocol stack. TCP sockets enable you to quickly developyour own custom client/server applications.“Communications and Networking,” is very useful with ell-established protocols, socketsallow you to develop your own modes of communication.

Sockets, as a programming interface, were originally developed at the University of California at Berkeley as a tool to easily accomplish network programming. Originally part of UNIX operatingsystems, the concept of sockets has been incorporated into a wide variety of operatingenvironments, including Java.

What Is a Socket?:

A socket is a handle to a communications link over the network with another application. A TCP socket uses the TCP protocol, inheriting the behavior of that transport protocol.

Four pieces of information are needed to create a TCP socket:

The local system’s IP address
The TCP port number the local application is using
The remote system’s IP address
The TCP port number to which the remote application is responding

Sockets are often used in client/server applications. A centralized service waits for variousremote machines to request specific resources, handling each request as it arrives. For clients to know how to communicate with the server, standard application protocols are assigned wellknownports. On UNIX operating systems, ports below 1024 can only be bound by applicationswith super-user (for example, root) privileges; thus, for control, these well-known ports liewithin this range, by convention. Some well-known ports are shown in Table 8.1.

Table 8.1 Well-Known TCP Ports and Services

Port	Service
21	FTP
23	elnet
25	SMTP (Standard Mail Transfer Protocol)
79	Finger
80	HTTP

8.5 JAVA TCP SOCKET CLASSES

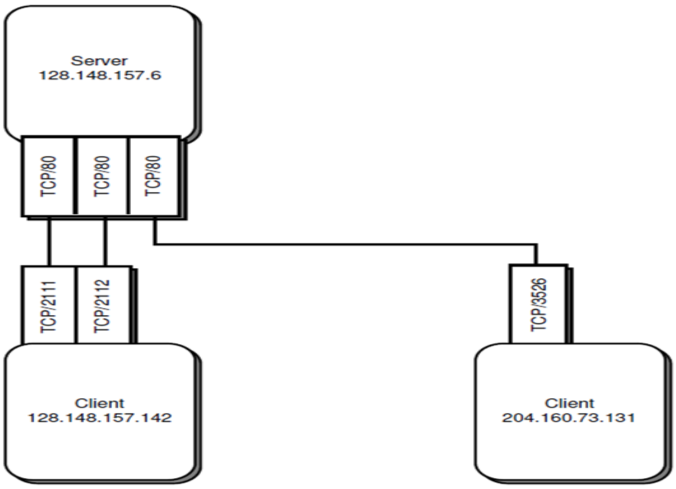
Java has a number of classes that allow you to create socket-based network applications. The two classes you use include java.net.Socket and java.net.ServerSocket.

T I P:

The Socket class is used for normal two-way socket communications and has four commonly used constructors:
public Socket(String host, int port)throws UnknownHostException,IOException;public Socket(InetAddress address, int port) throws IOException;public Socket(String host, int port, InetAddress localAddr, int localPort) throws UnknownHostException, IOException;public Socket(InetAddress address, int port, InetAddress localAddr, int localPort)throws UnknownHostException, IOException

The first constructor allows you to create a socket by just specifying the domain name of the remote machine within a String instance and the remote port. The second enables you tocreate a socket with an InetAddress object. The third and fourth are similar to the first two,except they allow you to choose the local interface and port number for the connection. If yourmachine has multiple IP addresses, you can use these constructors to choose a specific interfaceto use.

Figure 8.4.



An InetAddress is an object that stores an IP address of a remote system. It has no publicconstructor methods, but does have a number of static methods that return instances ofInetAddress. Thus, InetAddress objects can be created through static method invocations:

FIG. 8.5

Many clients can connect to a single server through separate sockets.

Server

128.148.157.6

Client

128.148.157.142

Client

204.160.73.131 TCP/80

TCP/80

TCP/80

TCP/2111

TCP/2112

TCP/3526

```
try {
InetAddress remoteOP =
InetAddress.getByName(“www.microsoft.com”);
InetAddress[] allRemoteIPs =
InetAddress.getAllByName(“www.microsoft.com”);
InetAddress myIP = InetAddress.getLocalHost();
} catch(UnknownHostException excpt) {
System.err.println(“Unknown host: “ + excpt);
}
```

The first method returns an InetAddress object with an IP address for www.microsoft.com.

The second obtains an array of InetAddress objects, one for each IP address mapped to www.microsoft.com.
The last InetAddress method creates an instance with the IP address ofthe local machine. All of these methods throw an UnknownHostException, which is caught in the previous example.

The Socket class has methods that allow you to read and write through the socket—the getInputStream() and getOutputStream() methods. To make applications simpler to design, the streams these methods return are usually decorated by another java.io object, such as BufferedReaderandPrintWriter, respectively. Both getInputStream() and getOutputStream() throw an IOException, which should be caught. Note the following:

```
try {
Socket netspace = new Socket(“www.netspace.org”,7);
BufferedReader input = new BufferedReader(
new InputStreamReader(netspace.getInputStream()));
PrintWriter output = new PrintWriter(
netspace.getOutputStream(), true);
} catch(UnknownHostException expt) {
System.err.println(“Unknown host: “ + excpt);
System.exit(1);
} catch(IOException excpt) {
System.err.println(“Failed I/O: “ + excpt);
System.exit(1);
}
```

To write a one-line message and then read a one-line response, you need only use the decorated

stream:

```
output.println(“test”);
String testResponse = input.readLine();
```

After you have completed communicating through the socket, you must first close the

InputStream and OutputStream instances, and then close the socket.

```
output.close();
```

```
input.close();
netspace.close();
```

To create a TCP server, it is necessary to understand a new class, ServerSocket. ServerSocket

allows you to bind a port and wait for clients to connect, setting up a complete Socket object at that time. ServerSocket has three constructors:

```
public ServerSocket(int port) throws IOException;
public ServerSocket(int port, int count)
 throws IOException;
public ServerSocket(int port, int count,
 InetAddress localAddr) throws IOException;
```

8.6 CREATING A TCP CLIENT/SERVER APPLICATION

The first constructor creates a listening socket at the port specified, allowing for the default number of 50 clients waiting in the connection queue. The second constructor enables you to change the length of the connection queue, allowing greater or fewer clients to wait to be processed by the server. The final constructor allows you to specify a local interface to listen for connections. If your machine has multiple IP addresses, this constructor allows you to provide services to specific IP addresses. Should you use the first two constructors on such a machine, the ServerSocket will accept connections to any of the machine’s IP addresses.

After creating a ServerSocket, the accept() method can be used to wait for a client to connect.

The accept() method blocks until a client connects, and then returns a Socket instance for communicating to the client. Blocking is a programming term that means a routine enters an internal loop indefinitely, returning only when a specific condition occurs. The program’s thread of execution does not proceed past the blocking routine until it returns—that is, when the specific condition happens.

The following code creates a ServerSocket at port 2222, accepts a connection, and then opens streams through which communication can take place once a client connects:

```
try {
ServerSocket server = new ServerSocket(2222);
Socket clientConn = server.accept();
BufferedReader input = new BufferedReader(
new InputStreamReader(clientConn.getInputStream()));
PrintWriter output = new PrintWriter(
clientConn.getInputStream(), true);
} catch(IOException excpt) {
System.err.println(“Failed I/O: “ + excpt);
System.exit(1);
}
```

Creating a TCP Client/Server Application

Designing an Application Protocol

Given the needs of our system, our protocol has six basic steps:

1. Client connects to server.
2. Server responds to client with a message indicating the currentness of the data.
3. The client requests data for a stock identifier.
4. The server responds.
5. Repeat steps 3 and 4 until the client ends the dialog.
6. Terminate the connection.

Implementing this design, you come up with a more detailed protocol. The server waits for the client on port 1701. When the client first connects, the server responds with:

```
+HELLO time-string
```

time-string indicates when the stock data to be returned was last updated. Next, the client sends a request for information. The server follows this by a response providing the data, as follows:

```
STOCK: stock-id
+stock-id stock-data
```

stock-id is a stock identifier consisting of a series of capital letters. stock-data is a string of characters detailing the performance of the particular stock. The client can request information on other stocks by repeating this request sequenceShould the client send a request for information regarding a stock of which the server is unaware, the server responds with:

```
-ERR UNKNOWN STOCK ID
```

If the client sends a command requesting information about a stock, but omits the stock ID, the server sends:

```
-ERR MALFORMED COMMAND
```

Should the client send an invalid command, the server responds with:

```
-ERR UNKNOWN COMMAND
```

When the client is done requesting information, it ends the communication and the server confirms the end of the session:

```
QUIT
+BYE
```

The next example demonstrates a conversation using the following protocol. All server responses should be preceded by a + or - character, while the client requests should not. In this example, the client is requesting information on three stocks: ABC, XYZ, and AAM. The serverhas information only regarding the last two:

```
+HELLO Tue, Jul 16, 1996 09:15:13 PDT
STOCK: ABC
-ERR UNKNOWN STOCK ID
STOCK: XYZ
+XYZ Last: 20 7/8; Change -0 1/4; Volume 60,400
STOCK: AAM
+AAM Last 35; Change 0; Volume 2,500
QUIT
+BYE
```

Developing the Stock Client

The client application to implement the preceding protocol should be fairly simple. The code is shown in Listing 8.1.

Listing 8.1 StockQuoteClient.java

```
import java.io.*; // Import the names of the packages
import java.net.*; // to be used.
/**
* This is an application which obtains stock information
* using our new application protocol.
*/
public class StockQuoteClient {
// The Stock Quote server listens at this port.
private static final int SERVER_PORT = 1701;
// Should your quoteSend PrintWriter autoflush?
private static final boolean AUTOFLUSH = true;
private String serverName;
private Socket quoteSocket = null;
private BufferedReader quoteReceive = null;
private PrintWriter quoteSend = null;
private String[] stockIDs; // Array of requested IDs.
private String[] stockInfo; // Array of returned data.
private String currentAsOf = null; // Timestamp of data.
/**
* Start the application running, first checking the
* arguments, then instantiating a StockQuoteClient, and
* finally telling the instance to print out its data.
* @param args Arguments which should be <server><stock ids>
*/
```

```
public static void main(String[] args) {
if (args.length < 2) {
System.out.println(
“Usage: StockQuoteClient <server><stock ids>”);
System.exit(1);
}
StockQuoteClient client = new StockQuoteClient(args);
client.printQuotes(System.out);
System.exit(0);
}
/**
* This constructor manages the retrieval of the
* stock information.
* @param args The server followed by the stock IDs.
*/
public StockQuoteClient(String[] args) {
String serverInfo;
// Server name is the first argument.
serverName = args[0];
// Create arrays as long as arguments - 1.
stockIDs = new String[args.length-1];
stockInfo = new String[args.length-1];
// Copy the rest of the elements of the args array
// into the stockIDs array.
for (int index = 1; index < args.length; index++) {
stockIDs[index-1] = args[index];
}
// Contact the server and return the HELLO message.
serverInfo = contactServer();
// Parse out the timestamp, which is everything after
// the first space.
if (serverInfo != null) {
currentAsOf = serverInfo.substring(
serverInfo.indexOf(“ “)+1);
}
getQuotes(); // Go get the quotes.
quitServer(); // Close the communication.
}
/**
* Open the initial connection to the server.
* @return The initial connection response.
*/
protected String contactServer() {
```

```
String serverWelcome = null;
try {
// Open a socket to the server.
quoteSocket = new Socket(serverName,SERVER_PORT);
// Obtain decorated I/O streams.
quoteReceive = new BufferedReader(
new InputStreamReader(
quoteSocket.getInputStream()));
quoteSend = new PrintWriter(
quoteSocket.getOutputStream(),
AUTOFLUSH);
// Read the HELLO message.
serverWelcome = quoteReceive.readLine();
} catch (UnknownHostException excpt) {
System.err.println(“Unknown host “ + serverName +
“: “ + excpt);
} catch (IOException excpt) {
System.err.println(“Failed I/O to “ + serverName +
“: “ + excpt);
}
return serverWelcome; // Return the HELLO message.
}
/**
* This method asks for all of the stock info.
*/
protected void getQuotes() {
String response; // Hold the response to stock query.
// If the connection is still up.
if (connectOK()) {
try {
// Iterate through all of the stocks.
for (int index = 0; index < stockIDs.length;
index++) {
// Send query.
quoteSend.println(“STOCK: “+stockIDs[index]);
// Read response.
response = quoteReceive.readLine();
// Parse out data.
stockInfo[index] = response.substring(
response.indexOf(“ “)+1);
}
} catch (IOException excpt) {
System.err.println(“Failed I/O to “ + serverName
+ “: “ + excpt);
}
}
}
/**
* This method disconnects from the server.
* @return The final message from the server.
*/
protected String quitServer() {
String serverBye = null; // BYE message.
try {
// If the connection is up, send a QUIT message
// and receive the BYE response.
if (connectOK()) {
quoteSend.println(“QUIT”);
serverBye = quoteReceive.readLine();
}
// Close the streams and the socket if the
// references are not null.
if (quoteSend != null) quoteSend.close();
if (quoteReceive != null) quoteReceive.close();
if (quoteSocket != null) quoteSocket.close();
} catch (IOException excpt) {
System.err.println(“Failed I/O to server “ +
serverName + “: “ + excpt);
}
return serverBye; // The BYE message.
}
/**
* This method prints out a report on the various
* requested stocks.
* @param sendOutput Where to send output.
*/
public void printQuotes(PrintStream sendOutput) {
// Provided that you actually received a HELLO message:
if (currentAsOf != null) {
sendOutput.print(“INFORMATION ON REQUESTED QUOTES”
+ “\n\tCurrent As Of: “ + currentAsOf + “\n\n”);
// Iterate through the array of stocks.
for (int index = 0; index < stockIDs.length;
index++) {
sendOutput.print(stockIDs[index] + “:”);
if (stockInfo[index] != null)
```

```
sendOutput.println(“ “ + stockInfo[index]);
else sendOutput.println();
}
}
}
/**
* Conveniently determine if the socket and streams are
* not null.
* @return If the connection is OK.
*/
protected boolean connectOK() {
return (quoteSend != null && quoteReceive != null &&
quoteSocket != null);
}
}
```

The main() Method: Starting the Client The main()method first checks to see that theapplication has been invoked with appropriate command-line arguments, quitting if this is notthe case. It then instantiates a StockQuoteClient with the args array reference and runs theprintQuotes() method, telling the client to send its data to standard output.

The StockQuoteClient Constructor The goal of the constructor is to initialize the data structures, connect to the server, load the stock data from the server, and terminate the connection. The constructor creates two arrays, one into which it copies the stock IDs and the other which remains uninitialized to hold the data for each stock.

It uses the contactServer() method to open communications with the server, returning the opening string. Provided the connection opened properly, this string contains a timestamp indicating the currentness of the stock data. The constructor parses this string to isolate that timestamp, gets the stock data with the getQuotes() method, and then closes the connection with quitServer().

The contactServer() Method: Starting the Communication Like the examples seen previously in this chapter, this method opens a socket to the server. It then creates two streams tocommunicate with the server. Finally, it receives the opening line from the server (for example, +HELLO time-string) and returns that as a String.

The getQuotes() Method: Obtaining the Stock Data This method performs the queries on each stock ID with which the application is invoked, now stored within the stockIDs array.

First it calls a short method, connectOK(), which merely ensures that the Socket and streams are not null. It iterates through the stockIDs array, sending each in a request to the server. It reads each response, parsing out the stock data from the line returned. It stores the stock data as a separate element in the stockInfo array. After it has requested information on each stock, the getQuotes() method returns.

The quitServer() Method: Ending the Connection This method ends the communicationwith the server, first sending a QUIT message if the connection is still valid. Then it performsthe essential steps when terminating a socket communication: it closes the streams and thenthe Socket.

The printQuotes() Method: Displaying the Stock Quotes Given a PrintStream object, suchas System.out, this method prints the stock data. It iterates through the array of stock identifiers,stockIDs, and then prints the value in the corresponding stockInfo array.

Developing the Stock Quote Server:

The server application is a bit more complex than the client that requests its services. It actually consists of two classes. The first loads the stock data and waits for incoming client connections.

When a client does connect, it creates an instance of another class that implements theRunnable interface, passing the newly created Socket to the client.

This secondary object, a handler, is run in its own thread of execution. This allows the serverto loop back and accept more clients, rather than perform the communications with clients oneat a time. When a server handles requests one after the other, it is said to be iterative; one thatdeals with multiple requests at the same time is concurrent. For TCP client/server interactions,which can often last a long time, concurrent operation is often essential. The handler isthe object that performs the actual communication with the client, and multiple instances of thehandler allow the server to process multiple requests simultaneously.This is a common network server design—using a multi-threaded server to allow many clientconnects to be handled simultaneously. The code for this application is shown in Listing 8.2.

Listing 8.2 StockQuoteServer.java

```
import java.io.*; // Import the package names to be
import java.net.*; // used by this application.
import java.util.*;
/**
* This is an application that implements our stock
* quote application protocol to provide stock quotes.
```

```
*/
public class StockQuoteServer {
// The port on which the server should listen.
private static final int SERVER_PORT = 1701;
// Queue length of incoming connections.
private static final int MAX_CLIENTS = 50;
// File that contains the stock data of format:
// <stock-id><stock information>
private static final File STOCK_QUOTES_FILE =
new File("stockquotes.txt");
private ServerSocket listenSocket = null;
private Hashtable stockInfo;
private Date stockInfoTime;
private long stockFileMod;
// A boolean used to keep the server looping until
// interrupted.
private boolean keepRunning = true;
/**
* Starts up the application.
* @param args Ignored command line arguments.
*/
public static void main(String[] args) {
StockQuoteServer server = new StockQuoteServer();
server.serveQuotes();
}
/**
* The constructor creates an instance of this class,
* loads the stock data, and then our server listens
* for incoming clients.
*/
public StockQuoteServer() {
// Load the quotes and exit if it is unable to do so.
if (!loadQuotes()) System.exit(1);
try {
// Create a listening socket.
listenSocket =
new ServerSocket(SERVER_PORT,MAX_CLIENTS);
} catch(IOException excpt) {
System.err.println("Unable to listen on port " +
SERVER_PORT + ": " + excpt);
System.exit(1);
}
}
```

```
/**
* This method loads in the stock data from a file.
*/
protected boolean loadQuotes() {
String fileLine;
StringTokenizer tokenize;
String id;
StringBuffer value;
try {
// Create a decorated stream to the data file.
BufferedReader stockInput = new BufferedReader(
new FileReader(STOCK_QUOTES_FILE));
// Create the Hashtable in which to place the data.
stockInfo = new Hashtable();
// Read in each line.
while ((fileLine = stockInput.readLine()) != null) {
// Break up the line into tokens.
tokenize = new StringTokenizer(fileLine);
try {
id = tokenize.nextToken();
// Ensure the stock ID is stored in upper case.
id = id.toUpperCase();
// Now create a buffer to place the stock value in.
value = new StringBuffer();
// Loop through all remaining tokens, placing them
// into the buffer.
while(tokenize.hasMoreTokens()) {
value.append(tokenize.nextToken());
// If there are more tokens to come, then append
// a space.
if (tokenize.hasMoreTokens()) {
value.append(“ “);
}
}
// Create an entry in our Hashtable.
stockInfo.put(id,value.toString());
} catch(NullPointerException excpt) {
System.err.println(“Error creating stock data “ +
“entry: “ + excpt);
} catch(NoSuchElementException excpt) {
System.err.println(“Invalid stock data record “ +
“in file: “ + excpt);
}
}
stockInput.close();
// Store the last modified timestamp.
stockFileMod = STOCK_QUOTES_FILE.lastModified();
} catch(FileNotFoundException excpt) {
System.err.println(“Unable to find file: “ + excpt);
return false;
} catch(IOException excpt) {
System.err.println(“Failed I/O: “ + excpt);
return false;
}
stockInfoTime = new Date(); // Store the time loaded.
return true;
}
/**
* This method waits to accept incoming client
* connections.
*/
public void serveQuotes() {
Socket clientSocket = null;
try {
while(keepRunning) {
// Accept a new client.
clientSocket = listenSocket.accept();
// Ensure that the data file hasn’t changed; if
// so, reload it.
if (stockFileMod !=
STOCK_QUOTES_FILE.lastModified()) {
loadQuotes();
}
// Create a new handler.
StockQuoteHandler newHandler = new
StockQuoteHandler(clientSocket,stockInfo,
stockInfoTime);
Thread newHandlerThread = new Thread(newHandler);
newHandlerThread.start();
}
listenSocket.close();
} catch(IOException excpt) {
System.err.println(“Failed I/O: “+ excpt);
}
}
/**
```

```
* This method allows the server to be stopped.
*/
protected void stop() {
if (keepRunning) {
keepRunning = false;
}
}
}
/**
* This class is used to manage a connection to
* a specific client.
*/
class StockQuoteHandler implements Runnable {
private static final boolean AUTOFLUSH = true;
private Socket mySocket = null;
private PrintWriter clientSend = null;
private BufferedReader clientReceive = null;
private Hashtable stockInfo;
private Date stockInfoTime;
/**
* The constructor sets up the necessary instance
* variables.
* @param newSocket Socket to the incoming client.
* @param info The stock data.
* @param time The time when the data was loaded.
*/
public StockQuoteHandler(Socket newSocket,
Hashtable info, Date time) {
mySocket = newSocket;
stockInfo = info;
stockInfoTime = time;
}
/**
* This is the thread of execution that implements
* the communication.
*/
public void run() {
String nextLine;
StringTokenizer tokens;
String command;
String quoteID;
String quoteResponse;
try {
```

```
clientSend =
new PrintWriter(mySocket.getOutputStream(),
AUTOFLUSH);
clientReceive =
new BufferedReader(new InputStreamReader(
mySocket.getInputStream()));
clientSend.println(“+HELLO “+ stockInfoTime);
// Read in a line from the client and respond.
while((nextLine = clientReceive.readLine())
!= null) {
// Break the line into tokens.
tokens = new StringTokenizer(nextLine);
try {
command = tokens.nextToken();
// QUIT command.
if (command.equalsIgnoreCase(“QUIT”)) break;
// STOCK command.
else if (command.equalsIgnoreCase(“STOCK:”)) {
quoteID = tokens.nextToken();
quoteResponse = getQuote(quoteID);
clientSend.println(quoteResponse);
}
// Unknown command.
else {
clientSend.println(“-ERR UNKNOWN COMMAND”);
}
} catch(NoSuchElementException excpt) {
clientSend.println(“-ERR MALFORMED COMMAND”);
}
}
clientSend.println(“+BYE”);
} catch(IOException excpt) {
System.err.println(“Failed I/O: “ + excpt);
// Finally close the streams and socket.
} finally {
try {
if (clientSend != null) clientSend.close();
if (clientReceive != null) clientReceive.close();
if (mySocket != null) mySocket.close();
} catch(IOException excpt) {
System.err.println(“Failed I/O: “ + excpt);
}
}
```

```
}
/**
* This method matches a stock ID to relevant information.
* @param quoteID The stock ID to look up.
* @return The releveant data.
*/
protected String getQuote(String quoteID) {
String info;
// Make sure the quote ID is in upper case.
quoteID = quoteID.toUpperCase();
// Try to retrieve from out Hashtable.
info = (String)stockInfo.get(quoteID);
// If there was such a key in the Hashtable, info will
// not be null.
if (info != null) {
return “+” + quoteID + “ “ + info;
}
else {
// Otherwise, this is an unknown ID.
return “-ERR UNKNOWN STOCK ID”;
}
}
}
```

Starting the Server The main() method allows the server to be started as an application andinstantiates a new StockQuoteServer object. It then uses the serveQuotes() method to beginaccepting client connections.The constructor first calls the loadQuotes() method to load in the stock data. The constructorensures that this process succeeds, and if not, quits the application. Otherwise, it creates aServerSocket at port 1701. Now the server is waiting for incoming clients.

The loadQuotes() Method: Read in the Stock Data This method uses a java.io.File objectto obtain a DataInputStream, reading in from the data file called “stockquotes.txt”.loadQuotes() goes through each line of the file, expecting that each line corresponds to a newstock with a format of:

Stock-ID stock-data:

The method parses the line and places the data into a Hashtable instance; the uppercase valueof the stock ID is the key while the stock data is the value. It stores the file’s modification timewith the lastModified() method of the File class, so the server can detect when the data hasbeen updated. It stores the current date using the java.util.Date class, so it can tell connectingclients when the stock information was loaded.In a more ideal design, this method would read data from the actual source of the stock information.Because you probably haven’t set up such a service within another company yet, astatic file will do for now.

The serveQuotes() Method: Respond to Incoming Clients This method runs in an infiniteloop, setting up connections to clients as they come in. It blocks at the accept() method of theServerSocket, waiting for a client to connect. When this occurs, it checks to see if the file inwhich the stock data resides has a different modification time since it was last loaded. If this is the case, it calls the loadQuotes() method to reload the data.

The serveQuotes() method then creates a StockQuoteHandler instance, passing it the Socketcreated when the client connected and the Hashtable of stock data. It places this handlerwithin a Thread object and starts that thread’s execution. After this has been performed, theserveQuotes() method loops back again to wait for a new client to connect.

Creating the StockQuotesHandler This class implements the Runnable interface so it canrun within its own thread of execution. The constructor merely sets some instance variables torefer to the Socket and stock data passed to it.

The run() Method: Implementing the Communication This method opens two streams toread from and write to the client. It sends the opening message to the client and then readseach request from the client. The method uses a StringTokenizer to parse the request andtries to match it with one of the two supported commands, STOCK: and QUIT.

If the request is a STOCK: command, it assumes the token after STOCK: is the stock identifierand passes the identifier to the getQuote() method to obtain the appropriate data. getQuote()is a simple method that tries to find a match within the stockInfo Hashtable. If one is found, itreturns the line. Otherwise, it returns an error message. The run() method sends this informationto the client.

If the request is a QUIT command, the server sends the +BYE response and breaks from theloop. It then terminates the communication by closing the streams and the Socket. The run()method ends, allowing the thread in which this object executes to terminate.Should the request be neither of these two commands, the server sends back an error message,waiting for the client to respond with a valid command.

Running the Client and Server:

Compile the two applications with javac. Then make sure you’ve created the stock quote datafile stockquotes.txt, as specified within the server code, in the proper format. Run the serverwith the Java interpreter, and it will run until interrupted by the system.

Finally, run the client to see how your server responds. Try running the client with one ormore of the stock identifiers you placed into the data file. Then, update the data file and tryyour queries again; the client should show that the data has changed.

8.7 OVERVIEW OF UDP MESSAGING

For many Internet developers, UDP (User Datagram Protocol) is used much less often thanTCP. UDP does not isolate you as neatly from the details of implementing a continuous networkcommunication. For many Java applications, however, choosing UDP as the tool to create anetwork linkage may be the most prudent option.Programming with UDP has significant ramifications. Understanding these factors will guideand educate your network programming efforts.UDP is a good choice for applications in which communications can be separated into discretemessages, where a single query from a client invokes a single response from a server. Time dependentdata is particularly suited to UDP. UDP requires much less overhead, but the burdenof engineering any necessary reliability into the system is your responsibility. For instance,if clients never receive responses to their queries—perfectly possible and legitimate withUDP—you might want to program the clients to retransmit the request or perhaps display an informative message indicating communication difficulties.

UDP Socket Characteristics:

As discussed in Chapter“Communications and Networking,” UDP behaves very differently than TCP. UDP is described as unreliable, connectionless, and message-oriented. Acommon analogy that explains UDP is that of communicating with postcards. A dialog with UDP must be quantified into small messages that fit within a small packet of aspecific size, although some packets can hold more data than others. When you send out a message, you can never be certain that you will receive a return message. Unless you do receivea return message, you have no idea if your message was received—your message could have been lost en route, the recipient’s confirmation could have been lost, or the recipient might be ignoring your message.The postcards you will be exchanging between network programs are referred to as*datagrams*. Within a datagram, you can store an array of bytes. A receiving application canextract this array and decode your message, possibly sending a return datagram response.As with TCP, you program in UDP using the socket programming abstraction. However, UDPsockets are very different from TCP sockets.

Extending the postcard analogy, UDP sockets aremuch like creating a mailbox.A mailbox is identified by your address, but you don’t construct a new one for each person towhom you will be sending a message. (However, you might create a new mailbox to receivenewspapers, which shouldn’t go into your normal mailbox.) Instead, you place an address onthe postcard that indicates to whom the message is being sent. You place the postcard in themailbox, and it is (eventually) sent on its way.

When receiving a message, you could potentially wait forever until one arrives in your mailbox.

After one arrives, you can read the postcard. Metainformation appears on the postcard thatidentifies the sender through the return address.As the previous analogies suggest, UDP programming involves the following general tasks:

- Creating an appropriately addressed datagram to send
- Setting up a socket to send and receive datagrams for a particular application
- Inserting datagrams into a socket for transmission
- Waiting to receive datagrams from a socket
- Decoding a datagram to extract the message, its recipient, and other meta information

Java UDP Classes:

The java.net package has the tools that are necessary to perform UDP communications. Forcreating datagrams, Java provides the DatagramPacket class. When receiving a UDP datagram,

you also use the DatagramPacket class to read the data, sender, and meta information.To create a datagram to send to a remote system, the following constructor is provided:
public DatagramPacket(byte[] ibuf, int length,InetAddress iaddr, int iport);

ibuf is the array of bytes that encodes the data of the message, while length is the length ofthe byte array to place into the datagram. This factor determines the size of the datagram.iaddr is an InetAddress object, which stores the IP address of the intended recipient. Portidentifies which port the datagram should be sent to on the receiving host.

See “Java TCP Socket Classes,”

To receive a datagram, you must use another DatagramPacket constructor in which the incoming data will be stored. This constructor has the prototype ofpublic DatagramPacket(byte[] ibuff, int ilength);ibuf is the byte array into which the data portion of the datagram will be copied.

ilength is thenumber of bytes to copy from the datagram into the array corresponding to the size of thedatagram. If ilength is less than the size of the UDP datagram received by the machine, theextra bytes will be silently ignored by Java.

According to the TCP/IP specification, the largest datagram possible is one that contains 65,507 bytesof data. However, a host is only required to receive datagrams with up to 548 bytes of data. Mostplatforms support larger datagrams of at least 8,192 bytes in length.Large datagrams are likely to be fragmented at the IP layer. If, during transmission, any one of the IP packets that contains a fragment of the datagram is lost, the entire UDP datagram will be silently lost.The point is you must design your application with the datagram size in mind. It is prudent to limit thissize to a reasonable length.
After a datagram has been received, as illustrated later in this section, you can read that data.

Other methods allow you to obtain meta information regarding the message:

```
public int getLength();
public byte[] getData();
public InetAddress getAddress();
public int getPort();
```

The getLength() method is used to obtain the number of bytes contained within the data portion of the datagram. The getData() method is used to obtain a byte array containing the data received. getAddress() provides an InetAddress object identifying the sender, while getPort() indicates the UDP port used.

Performing the sending and receiving of these datagrams is accomplished with theDatagramSocket class, which creates a UDP socket. Three constructors are available:

```
public DatagramSocket() throws IOException;
public DatagramSocket(int port) throws IOException;
public DatagramSocket(int port, InetAddress localAddr)
 throws IOException
;
```

The first constructor allows you to create a socket at an unused ephemeral port, generally usedfor client applications. The second constructor allows you to specify a particular port, which isuseful for server applications. As with TCP, most systems require super-user privileges to bindUDP ports below 1024. The final constructor is useful for machines with multiple IP interfaces.You can use this constructor to send and listen for datagrams from one of the IP addressesassigned to the machine. On such a host, datagrams sent to any of the machine’s IP addressesare received by a DatagramSocket created with the first two constructors, while the last constructorobtains only datagrams sent to the specific IP address.You can use this socket to send properly addressed DatagramPacket instances created with thefirst constructor described by using this DatagramSocket method:

```
public void send(DatagramPacket p) throws IOException;
```

After a DatagramPacket has been created with the second constructor described, a datagram can be received:

```
public synchronized void receive(DatagramPacket p)
 throws IOException;
```

Note that the receive() method blocks until a datagram is received. Because UDP is unreliable, your application cannot expect receive() ever to return unless a timeout is enabled. Sucha timeout, named the SO_TIMEOUT option from the name of the Berkeley sockets API option,can be set with this method from the DatagramSocket class:

public synchronized void setSoTimeout(int timeout)
throws SocketException;timeout is a value in milliseconds. If set to 0, the receive() method exhibits an infinitetimeout—the default behavior. When greater than zero, a subsequent receive() method invocationwaits only the specified timeout before an InterruptedIOException is thrown.

8.8 CREATING A UDP SERVER

In this section, you learn how to create a basic UDP server that responds to simple clientrequests.The practical example used here is to create a daytime server.Daytime is a simple service that runs on many systems. For example, most UNIX systems rundaytime out of inetd, as listed in /etc/inetd.conf. On Windows NT, the daytime server isavailable through the Simple TCP/IP Services within the Services Control Panel. Daytime isgenerally run on UDP port 13. When sent a datagram, it responds with a datagram containingthe date in a format such as

Friday, July 30, 1993 19:25:00

Listing 8.3 shows the Java code used to implement this service.

Listing 8.3 DaytimeServer.java

```
import java.net.*; // Import the package names used
import java.util.*;
import java.io.*;
import java.text.*;
/**
* This is an application that runs the
* daytime service.
*/
public class DaytimeServer {
```

```
// The daytime service runs on this well known port.
private static final int TIME_PORT = 13;
private DatagramSocket timeSocket = null;
private static final int SMALL_ARRAY = 1;
private static final int TIME_ARRAY = 100;
// A boolean to keep the server looping until stopped.
private boolean keepRunning = true;
/**
* This method starts the application, creating an
* instance and telling it to start accepting
* requests.
* @param args Command line arguments - ignored.
*/
public static void main(String[] args) {
DaytimeServer server = new DaytimeServer();
server.startServing();
}
/**
* This constructor creates a datagram socket to
* listen on.
*/
public DaytimeServer() {
try {
timeSocket = new DatagramSocket(TIME_PORT);
} catch(SocketException excpt) {
System.err.println(“Unable to open socket: “ +
excpt);
}
}
/**
* This method does all of the work of listening for
* and responding to clients.
*/
public void startServing() {
DatagramPacket datagram; // For a UDP datagram.
InetAddress clientAddr; // Address of the client.
int clientPort; // Port of the client.
byte[] dataBuffer; // To construct a datagram.
String timeString; // The time as a string.
// Keep looping while you have a socket.
while(keepRunning) {
try {
// Create a DatagramPacket to receive query.
dataBuffer = new byte[SMALL_ARRAY];
datagram = new DatagramPacket(dataBuffer,
dataBuffer.length);
timeSocket.receive(datagram);
// Get the meta-info on the client.
clientAddr = datagram.getAddress();
clientPort = datagram.getPort();
// Place the time into byte array.
dataBuffer = getTimeBuffer();
// Create and send the datagram.
datagram = new DatagramPacket(dataBuffer,
dataBuffer.length,clientAddr,clientPort);
timeSocket.send(datagram);
} catch(IOException excpt) {
System.err.println(“Failed I/O: “ + excpt);
}
}
timeSocket.close();
}
* This method is used to create a byte array
* containing the current time in the special daytime
* server format.
* @return The byte array with the time.
*/
protected byte[] getTimeBuffer() {
String timeString;
SimpleDateFormat daytimeFormat;
Date currentTime;
// Get the current time.
currentTime = new Date();
// Create a SimpleDateFormat object with the time
// pattern specified.
// EEEE - print out complete text for day
// MMMM - print out complete text of month
// dd - print out the day in month in two digits
// yyyy - print out the year in four digits
// HH - print out the hour in the day, from 0-23
// in two digits
// mm - print out the minutes in the hour in two
// digits
// ss - print out the seconds in the minute in
// two digits
daytimeFormat =
new SimpleDateFormat(“EEEE, MMMM dd, yyyy HH:mm:ss”);
// Create the special time format.
timeString = daytimeFormat.format(currentTime);
// Convert the String to an array of bytes using the
// platform’s default character encoding.
return timeString.getBytes();
}
/**
* This method provides an interface to stopping
* the server.
*/
protected void stop() {
if (keepRunning) {
keepRunning = false;
}
}
/**
* Just in case, do some cleanup.
*/
public void finalize() {
if (timeSocket != null) {
timeSocket.close();
}
}
}
```

Starting the Server:

The DaytimeServer class uses a number of static final variables as constants, many of whichare used to create the date string in the proper format. The main() method creates aDaytimeServer object and then invokes its startServing() method so that it accepts incomingrequests.

The DaytimeServer constructor merely creates a UDP socket at the specified port. Note thatas written, the server may require super-user privileges to run because it binds port 13. If youdon’t have permission to bind this port, the attempt to create a DatagramSocket throws anexception. The constructor catches this and fails gracefully, informing you of the problem.The DaytimeServer is an iterative server, whereas the server created in Chapter , “TCPSockets,” is a concurrent server. DaytimeServer processes each request in serial as they comein. Given the nature of the protocol—a single datagram comes in and the server immediatelysends back a datagram with the time—an iterative server is most appropriate and is simpler toprogram.

The startServing() Method Handling Requests

The startServing() method is where the serving logic is implemented. Although the applicationis intended to be running, it loops through a number of steps: It creates a small byte arrayand uses this array to create a DatagramPacket. The application then receives a datagram fromthe DatagramSocket. From the datagram, it obtains the IP address and port of the requestingapplication. The startServing() method need not read any information from the incomingdatagram, as the datagram’s arrival plus the meta-information it contains is sufficient for theserver to understand the request.

The getTimeBuffer() method is called to obtain a byte array that contains the time in an appropriateformat. By using this information, this method creates a new DatagramPacket. Finally,it sends this information through the DatagramSocket. The server loops through this process until interrupted externally.

The getTimeBuffer() Method Creating the Byte Array

This protected method creates an instance of Date class containing the current time and then instantiates a SimpleDateFormat with a specific time pattern. It uses the SimpleDateFormat object to create a String with the data in the proper format. Finally, it returns the byte array corresponding to that String.

Running the Daytime Server:

To run the server, first compile it with javac. Then, if necessary, log in as the super user (forexample, “root”) and use java to run the server. If this is not possible, modify the TIME_PORT variable so that it binds to a port over 1024.

In the next example, you create a client to connect to this server.

8.9 CREATING A UDP CLIENT

The example used to create a UDP client makes use of the daytime server demonstrated previouslybut also illustrates communications with multiple servers through a single UDP socket.TimeCompare is a Java program that requests the time from a series of servers, receives theirresponses, and displays the difference between the remote system’s times and the time of thelocal machine.

One of the most important aspects of this client is designing it so that an unanswered querydoes not hang the program. You cannot expect that every query will be answered. Thus, youneed to use the setSoTimeout() method of the DatagramSocket instance before you call receive().

Listing 8.4 shows this application.

Listing 8.4 TimeCompare.java

```
import java.io.*; // Import the package names used.
import java.net.*;
import java.util.*;
import java.text.*;
/**
* This is an application to obtain the times from
* various remote systems via UDP and then report
* a comparison.
*/
public class TimeCompare {
private static final int TIME_PORT = 13; // Daytime port.
private static final int TIMEOUT = 10000; // UDP timeout.
// This is the size of the datagram data to send
// for the query - intentially small.
private static final int SMALL_ARRAY = 1;
// This is the size of the datagram you expect to receive.
private static final int TIME_ARRAY = 100;
// A socket to send and receive datagrams.
DatagramSocket timeSocket = null;
// An array of addresses to the machines to query.
private InetAddress[] remoteMachines;
// The time on this machine.
private Date localTime;
// An array of datagram responses from remote machines.
private DatagramPacket[] timeResponses;
/**
* This method starts the application.
* @param args Command line arguments - remote hosts.
*/
public static void main(String[] args) {
if (args.length < 1) {
System.out.println(
“Usage: TimeCompare host1 (host2 ... hostn)”);
System.exit(1);
}
```

```
// Create an instance.
TimeCompare runCompare = new TimeCompare(args);
// Tell it to print out its data.
runCompare.printTimes();
System.exit(0); // Exit.
}
/**
* The constructor looks up the remote hosts and
* creates a UDP socket.
* @param hosts The hosts to contact.
*/
public TimeCompare(String[] hosts) {
remoteMachines = new InetAddress[hosts.length];
// Look up all hosts and place in InetAddress[] array.
for(int hostsFound = 0; hostsFound < hosts.length;
hostsFound++) {
try {
remoteMachines[hostsFound] =
InetAddress.getByName(hosts[hostsFound]);
} catch(UnknownHostException excpt) {
remoteMachines[hostsFound] = null;
System.err.println("Unknown host " +
hosts[hostsFound] + ": " + excpt);
}
}
try {
timeSocket = new DatagramSocket();
} catch(SocketException excpt) {
System.err.println("Unable to bind UDP socket: " +
excpt);
System.exit(1);
}
// Perform the UDP communications.
getTimes();
}
/**
* This method is the thread of execution where you
* send out requests for times and then receive the
* responses.
*/
public void getTimes() {
DatagramPacket timeQuery; // A datagram to send as a query.
DatagramPacket response; // A datagram response.
```

```
byte[] emptyBuffer; // A byte array to build datagrams.
int datagramsSent = 0; // # of queries successfully sent.
// Send out a small UDP datagram to each machine,
// asking it to respond with its time.
for(int ips = 0;ips < remoteMachines.length; ips++) {
if (remoteMachines[ips] != null) {
try {
emptyBuffer = new byte[SMALL_ARRAY];
timeQuery = new DatagramPacket(emptyBuffer,
emptyBuffer.length, remoteMachines[ips],
TIME_PORT);
timeSocket.send(timeQuery);
datagramsSent++;
} catch(IOException excpt) {
System.err.println(“Unable to send to “ +
remoteMachines[ips] + “: “ + excpt);
}
}
}
// Get current time to base the comparisons.
localTime = new Date();
// Create an array in which to place responses.
timeResponses = new DatagramPacket[datagramsSent];
// Set the socket timeout value.
try {
timeSocket.setSoTimeout(TIMEOUT);
} catch(SocketException e) {}
// Loop through and receive the number of responses
// you are expecting. You break from this loop prematurely
// if an InterruptedIOException occurs - that is, if
// you wait more than TIMEOUT to receive another datagram.
try {
for(int got = 0; got < timeResponses.length; got++) {
// Create a new buffer and datagram.
emptyBuffer = new byte[TIME_ARRAY];
response = new DatagramPacket(emptyBuffer,
emptyBuffer.length);
// Receive a datagram, timing out if necessary.
timeSocket.receive(response);
// Now that you’ve received a response, add it
// to the array of received datagrams.
timeResponses[got] = response;
}
```

```
} catch(InterruptedIOException excpt) {
System.err.println("Timeout on receive: " + excpt);
} catch(IOException excpt) {
System.err.println("Failed I/O: " + excpt);
}
// Close the socket.
timeSocket.close();
timeSocket = null;
}
/**
* This prints out a report comparing the times
* sent from the remote hosts with the local
* time.
*/
protected void printTimes() {
Date remoteTime;
String timeString;
long secondsOff;
InetAddress dgAddr;
SimpleDateFormat daytimeFormat;
System.out.print("TIME COMPARISON\n\tCurrent time " +
"is: " + localTime + "\n\n");
// Iterate through each host.
for(int hosts = 0;
hosts < remoteMachines.length; hosts++) {
if (remoteMachines[hosts] != null) {
boolean found = false;
int dataIndex;
// Iterate through each datagram received.
for(dataIndex = 0; dataIndex < timeResponses.length;
dataIndex++) {
// If the datagram element isn't null:
if (timeResponses[dataIndex] != null) {
dgAddr = timeResponses[dataIndex].getAddress();
// See if there's a match.
if(dgAddr.equals(remoteMachines[hosts])) {
found = true;
break;
}
}
}
System.out.println('Host: ' +
remoteMachines[hosts]);
// If there was a match, print comparison.
if (found) {
timeString =
new String(timeResponses[dataIndex].getData());
int endOfLine = timeString.indexOf(“\n”);
if (endOfLine != -1) {
timeString =
timeString.substring(0,endOfLine);
}
// Create a SimpleDateFormat object with the time
// pattern specified.
// EEEE - print out complete text for day
// MMMM - print out complete text of month
// dd - print out the day in month in two digits
// yyyy - print out the year in four digits
// HH - print out the hour in the day, from 0-23
// in two digits
// mm - print out the minutes in the hour in two
// digits
// ss - print out the seconds in the minute in
// two digits
daytimeFormat =
new SimpleDateFormat(“EEEE, MMMM dd, yyyy HH:mm:ss”);
// Parse the string based on the pattern into a
// Date object.
remoteTime = daytimeFormat.parse(timeString,
new ParseStatus());
// Find the difference.
secondsOff = (localTime.getTime() -
remoteTime.getTime()) / 1000;
secondsOff = Math.abs(secondsOff);
System.out.println(“Time: “ + timeString);
System.out.println(“Difference: “ +
secondsOff + “ seconds\n”);
} else {
System.out.println(“Time: NO RESPONSE FROM “
+ “HOST\n”);
}
}
}
}
/**
* This method performs any necessary cleanup.
```

```
*/
protected void finalize() {
// If the socket is still open, close it.
if (timeSocket != null) {
timeSocket.close();
}
}
}
```

Starting TimeCompare :

The main() method instantiates a TimeCompare object, passing it the command-line argumentsthat correspond to the hosts to query. main() instructs the instance to print out its data andthen exits.The TimeCompare constructor uses the InetAddress.getByName() static method to look upthe set of remote hosts, placing the returned InetAddress instances into an array of theseobjects. If it is unable to look up one of the hosts, this constructor ensures that the element isset to null and loops through the other hosts. The constructor creates a DatagramSocket at adynamically allocated port and finally calls the getTimes() method to perform the queries.

The getTimes() Method TimeCompare's Execution PathThe first thing this method does is iterate through the remoteMachines array. For each elementthat is not null, the getTimes() method creates a small byte array, uses it to construct anappropriatelyaddressed DatagramPacket, and then sends the datagram using the UDP socket.After it's sent, the method uses datagramsSent to keep track of how many queries were successfullysent.

Now that a datagram has been sent to each remote host, TimeCompare prepares to receive the responses. At this point, getTimes() collects the current time, used as a basis for comparisonagainst the remote systems' times. It creates an array of type DatagramPacket. The length ofthis array is equal to the number of successful queries sent, which is the number of expectedresponses. getTimes() then invokes the setSoTimeout() method of the DatagramSocket instanceso that accept() will not block forever if a server fails to respond.getTimes() next enters a loop, attempting to receive a DatagramPacket for each query successfullysent. When a response is obtained, it places it into the timeResponses array. If the timeouton the receive() method expires, it breaks out of the loop.

After getTimes() has completed the loop to receive responses, it closes the DatagramSocket.

The printTimes() Method Showing the ComparisonThis method takes an array of UDP packets and prints out a comparison of the times containedtherein. The outer for loop iterates through the machines contacted, while the inner for loopmatches the host to a received datagram. If a match is found, printTimes() calculates thedifference in times and prints the data. If no match is found, printTimes() indicates that aresponse from that host was not received.

Running the Application:

Compile TimeCompare.java with the Java compiler and then execute it with the Java interpreter.

Each argument to TimeCompare should be a host name of a remote machine to includein the comparison. For instance, to check your machine’s time against www.sgi.com andwww.paramount.com, you would typejava TimeCompare www.sgi.com www.paramount.com and you would see a report that appeared as

```
TIME COMPARISON
Current time is: Mon Aug 19 08:03:09 PDT 1996
Host: www.sgi.com/204.94.214.4
Time: Mon Aug 19 08:02:55 1996
Difference: 14 seconds
Host: www.paramount.com/192.216.189.10
Time: Mon Aug 19 08:07:58 1996
Difference: 288 seconds
```

8.10 USING IP MULTICASTING

Internet Protocol (IP) is the means by which all information on the Internet is transmitted.UDP datagrams are encapsulated within IP packets to send them to the appropriate machineson the network.Most uses of IP involve unicasting—sending a packet from one host to another. However, IPis not limited to this mode and includes the capability to multicast. With multicasting, a messageis addressed to a targeted set of hosts. One message is sent, and the entire group canreceive it.

Multicasting is particularly suited to high-bandwidth applications, such as sending video andaudio over the network, because a separate transmission need not be established (which couldsaturate the network). Other possible applications include chat sessions, distributed data storage,and online, interactive games. Also, multicasting can be used by a client searching for anappropriate server on the network; it can send a multicast solicitation, and any listening serverscould contact the client to begin a transaction.To support IP multicasting, a certain range of IP addresses is set aside solely for this purpose.These IP addresses are class D addresses, those within the range of

224.0.0.0 and239.255.255.255. Each of these addresses is referred to as a *multicast group*. Any IP packetaddressed to that group is received by any machine that has joined that group. Group membershipis dynamic and changes over time. To send a message to a group, a host need not be amember of that group.

When a machine joins a multicast group, it begins accepting messages sent to that IP multicastaddress. Extending the previous analogy from the section “UDP Socket Characteristics,” joininga group is similar to constructing a new mailbox that accepts messages intended for thegroup. Each machine that wants to join the group constructs its own mailbox to receive thesame message. If a multicast packet is distributed to a network, any machine that is listeningfor the message has an opportunity to receive it. That is, with IP multicasting, there is nomechanism for restricting which machines on the same network may join the group.Multicast groups are mapped to hardware addresses on interface cards. Thus, IP multicastdatagrams that reach an uninterested host can usually be rapidly discarded by the interfacecard. However, more than one multicast group maps to a single hardware address, making forimperfect hardware-level filtering. Some filtering must still be performed at the device driveror IP level.

Multicasting has its limitations, however—particularly the task of routing multicast packetsthroughout the Internet. A special TCP/IP protocol, Internet Group Management Protocol(IGMP), is used to manage memberships in a multicast group. A router that supports multicastingcan use IGMP to determine if local machines are subscribed to a particular group; suchhosts respond with a report about groups they have joined using IGMP. Based on these communications,a multicast router can determine if it is appropriate to forward on a multicast packet.Besides the multicast group, another important facet of a multicast packet is the time-to-live(TTL) parameter. The TTL is used to indicate how many separate networks the sender intendsthe message to be transmitted over. When a packet is forwarded on by a router, the TTL withinthe packet is decremented by one. When a TTL reaches zero, the packet is not forwarded on further.

The Multicast Backbone, or MBONE, is an attempt to create a network of Internet routers that are capable of providing multicast services. However, multicasting today is by no means ubiquitous.If all participants reside on the same physical network, routers need not be involved, andmulticasting is likely to prove successful. For more distributed communications, you may needto contact your network administrator.

Java Multicasting:

The Java MulticastSocket class is the key to utilizing this powerful Internet networking feature.

MulticastSocket allows you to send or receive UDP datagrams that use multicast IP. Tosend a datagram, you use the default constructor:

```
public MulticastSocket() throws IOException;
```

Then you must create an appropriately formed DatagramPacket addressed to a multicast group between 224.0.0.0 and 239.255.255.255. After it is created, the datagram can be sent with thesend() method, which requires a TTL value. The TTL indicates how many routers the packetsshould be allowed to go through. Avoid setting the TLL to a high value, which could cause thedata to propagate through a large portion of the Internet. Here is an example:

```
int multiPort = 2222;
int ttl = 1;
InetAddressmultiAddr
=InetAddress.getByName(“239.10.10.10”);
byte[] multiBytes = new byte[256];
DatagramPacket multiDatagram =new
DatagramPacket(multiBytes,
multiBytes.length,multiAddr,multiPort);
MulticastSocket multiSocket = new MulticastSocket();
 multiSocket.send(multiDatagram, ttl);
```

To receive datagrams, an application must create a socket at a specific UDP port. Then, it mustjoin the group of recipients. Through the socket, the application can then receive UDPdatagrams:

```
MulticastSocket receiveSocket = new
MulticastSocket(multiPort);
receiveSocket.joinGroup(multiAddr);
receiveSocket.receive(multiDatagram);
```

When the joinGroup() method is invoked, the machine now pays attention to any IP packetstransmitted along the network for that particular multicast group. The host should also useIGMP to appropriately report the usage of the group. For machines with multiple IP addresses,the interface through which datagrams should be sent can be configured:

```
receiveSocket.setInterface(oneOfMyLocalAddrs);
```

To leave a multicast group, the leaveGroup() method is available. A MulticastSocket should be closed when communications are done:

```
receiveSocket.leaveGroup(multiAddr);
receiveSocket.close();
```

As is apparent, using the MulticastSocket is very similar to using the normal UDP socketclass DatagramSocket. The essential differences are

- The DatagramPacket must be addressed to a multicast group.
- The send() method of the MulticastSocket class takes two arguments: aDatagramPacket and a TTL value.
- To begin listening for multicast messages, after creating the MulticastSocket instance,you must use the joinGroup() method.
- The receive() method is used just as with the DatagramSocket to obtain incomingmessages, though there is no method to set a timeout, like setSoTimeout() inDatagramSocket.

Multicast Applications :

The following two examples show a very simple use of multicasting. Listing 8.5 is a programthat sends datagrams to a specific multicast IP address. The program is run with two arguments:the first specifying the multicast IP address to send the datagrams, the other specifyingthe UDP port of the listening applications. The main() method ensures that these argumentshave been received and then instantiates a MultiCastSender object.The constructor creates an InetAddress instance with the String representation of themulticast IP address. It then creates a MulticastSocket at a dynamically allocated port forsending datagrams. The constructor enters a while loop, reading in from standard input line byline. The program packages the first 256 bytes of each line into an appropriately addressedDatagramPacket, sending that datagram through the MulticastSocket.

Listing 8.5 MultCastSender.java

```
import java.net.*; // Import package names used.
import java.io.*;
/**
* This is a program that sends data from the command
* line to a particular multicast group.
*/
class MultiCastSender {
// The number of Internet routers through which this
// message should be passed. Keep this low. 1 is good
// for local LAN communications.
private static final byte TTL = 1;
// The size of the data sent - basically the maximum
// length of each line typed in at a time.
private static final int DATAGRAM_BYTES = 512;
private int mcastPort;
private InetAddress mcastIP;
private BufferedReader input;
private MulticastSocket mcastSocket;
/**
* This starts up the application.
* @param args Program arguments - <ip><port>
*/
public static void main(String[] args) {
// This must be the same port and IP address used
// by the receivers.
if (args.length != 2) {
System.out.print("Usage: MultiCastSender <IP addr>"
+ " <port>\n\t<IP addr> can be one of 224.x.x.x "
+ "- 239.x.x.x\n");
System.exit(1);
}
MultiCastSender send = new MultiCastSender(args);
System.exit(0);
}
/**
* The constructor does all of the work of opening
* the socket and sending datagrams through it.
* @param args Program arguments - <ip><port>
*/
public MultiCastSender(String[] args) {
DatagramPacket mcastPacket; // UDP datagram.
String nextLine; // Line from STDIN.
byte[] mcastBuffer; // Buffer for datagram.
byte[] lineData; // The data typed in.
int sendLength; // Length of line.
input =
new BufferedReader(new InputStreamRea
mcastIP = InetAddress.getByName(args[0]);
mcastPort = Integer.parseInt(args[1]);
mcastSocket = new MulticastSocket();
} catch(UnknownHostException excpt) {
System.err.println("Unknown address: " + excpt);
System.exit(1);
} catch(IOException excpt) {
System.err.println("Unable to obtain socket: "
+ excpt);
System.exit(1);
}
try {
// Loop and read lines from standard input.
while ((nextLine = input.readLine()) != null) {
mcastBuffer = new byte[DATAGRAM_BYTES];
// If line is longer than your buffer, use the
// length of the buffer available.
if (nextLine.length() > mcastBuffer.length) {
sendLength = mcastBuffer.length;
// Otherwise, use the line’s length.
} else {
sendLength = nextLine.length();
}
// Convert the line of input to bytes.
lineData = nextLine.getBytes();
// Copy the data into the blank byte array
// which you will use to create the DatagramPacket.
for (int i = 0; i < sendLength; i++) {
mcastBuffer[i] = lineData[i];
}
mcastPacket = new DatagramPacket(mcastBuffer,
mcastBuffer.length,mcastIP,mcastPort);
// Send the datagram.
try {
System.out.println(“Sending:\t” + nextLine);
mcastSocket.send(mcastPacket,TTL);
} catch(IOException excpt) {
System.err.println(“Unable to send packet: “
+ excpt);
}
}
} catch(IOException excpt) {
System.err.println(“Failed I/O: “ + excpt);
}
mcastSocket.close(); // Close the socket.
}
}
```

Listing 8.5 complements the sender by receiving multicasted datagrams. The applicationtakes two arguments that must correspond to the IP address and port with which theMultiCastSender was invoked. The main() method checks the command-line arguments andthen creates a MultiCastReceiver object.The object’s constructor creates an InetAddress and then a MulticastSocket at the port usedto invoke the application. It joins the multicast group at the address contained within theInetAddress instance and then enters a loop. The object’s constructor receives a datagramfrom the socket and prints the data contained within the datagram, indicating the machine andport from where the packet was sent.

Listing 8.6 MultiCastReceiver.java

```
import java.net.*; // Import package names used.
import java.io.*;
/**
* This is a program that allows you to listen
* at a particular multicast IP address/port and
* print out incoming UDP datagrams.
*/
class MultiCastReceiver {
// The length of the data portion of incoming
// datagrams.
private static final int DATAGRAM_BYTES = 512;
private int mcastPort;
private InetAddress mcastIP;
private MulticastSocket mcastSocket;
// Boolean to tell the client to keep looping for
// new datagrams.
private boolean keepReceiving = true;
/**
* This starts up the application
* @param args Program arguments - <ip><port>
*/
public static void main(String[] args) {
// This must be the same port and IP address
// used by the sender.
if (args.length != 2) {
System.out.print(“Usage: MultiCastReceiver <IP “
+ “addr><port>\n\t<IP addr> can be one of “
+ “224.x.x.x - 239.x.x.x\n”);
System.exit(1);
}
MultiCastReceiver send = new MultiCastReceiver(args);
System.exit(0);
}
/**
* The constructor does the work of opening a socket,
* joining the multicast group, and printing out
* incoming data.
* @param args Program arguments - <ip><port>
*/
```

```
public MultiCastReceiver(String[] args) {
DatagramPacket mcastPacket; // Packet to receive.
byte[] mcastBuffer; // byte[] array buffer

InetAddress fromIP; // Sender address.
int fromPort; // Sender port.
String mcastMsg; // String of message.
try {
// First, set up your receiving socket.
mcastIP = InetAddress.getByName(args[0]);
mcastPort = Integer.parseInt(args[1]);
mcastSocket = new MulticastSocket(mcastPort);
// Join the multicast group.
mcastSocket.joinGroup(mcastIP);
} catch(UnknownHostException excpt) {
System.err.println(“Unknown address: “ + excpt);
System.exit(1);
} catch(IOException excpt) {
System.err.println(“Unable to obtain socket: “
+ excpt);
System.exit(1);
}
while (keepReceiving) {
try {
// Create a new datagram.
mcastBuffer = new byte[DATAGRAM_BYTES];
mcastPacket = new DatagramPacket(mcastBuffer,
mcastBuffer.length);
// Receive the datagram.
mcastSocket.receive(mcastPacket);
fromIP = mcastPacket.getAddress();
fromPort = mcastPacket.getPort();
mcastMsg = new String(mcastPacket.getData());
// Print out the data.
System.out.println(“Received from “ + fromIP +
“ on port “ + fromPort + “: “ + mcastMsg);
} catch(IOException excpt) {
System.err.println(“Failed I/O: “ + excpt);
}
}
try {
mcastSocket.leaveGroup(mcastIP); // Leave the group.
} catch(IOException excpt) {
System.err.println(“Socket problem leaving group: “
+ excpt);
}
mcastSocket.close(); // Close the socket.
}
/**
* This method provides a way to stop the program.
*/
public void stop() {
if (keepReceiving) {
keepReceiving = false;
}
}
}
```

To run the applications, first compile MultiCastSender and MultiCastReceiver. Then, transfertheMultCastReceiver to other machines, so you can demonstrate more than one participantreceiving messages. Finally, run the applications with the Java interpreter.For instance, to send ulticast messages to the group 224.0.1.30 on port 1111, you could do thefollowing:

```
~/classes -> java MultiCastSender 224.0.1.30 1111
This is a test multicast message.
Sending: This is a test multicast message.
Have you received it?
Sending: Have you received it?
```

To receive these messages, you would run the MultiCastReceiver application on one or moresystems. You join the same multicast group, 224.0.1.30, and listen to the same port number,

1111:

```
~/classes -> java MultiCastReceiver 224.0.1.30 1111
Received from 204.160.73.131 on port 32911: This is a test multicast message.
Received from 204.160.73.131 on port 32911: Have you received it?
```

8.11 THE URL CLASS

```
try {
URL myURL = new URL(getDocumentBase(), “foo.html”);
InputStream in = myURL.openStream(); // get input stream for URL
int b;
while ((b = in.read()) != -1) { // read the next byte
System.out.print((char)b); // print it
}
} catch (Exception e) {
e.printStackTrace(); // something went wrong
}
```

Getting URL Information

You can retrieve the specific pieces of a URL using the following methods:

public String getProtocol() returns the name of the URL’s protocol.

public String getHost() returns the name of the URL’s host.

public int getPort() returns the URL’s port number.

public String getFile() returns the URL’s filename.

public String getRef() returns the URL’s reference tag. This is an optional index into an HTML page that follows the filename and begins with a #.

8.12 THE URLCONNECTION CLASS

The URLConnection class provides a more granular interface to a URL than the getContentmethod in the URL class. This class provides methods for examining HTTP headers, gettinginformation about the URL’s content, and getting input and output streams to the URL. Therewill be a different URLConnection class for each type of protocol that you can use. There will bea URLConnection that handles the HTTP protocol, for example, as well as another that handlesthe FTP protocol. Your browser may not support any of them. You can feel fairly certain thatthey are implemented in HotJava. HotJava is written totally in Java, which uses these classes todo all of its browsing. Netscape, on the other hand, has its own native code for handling theseprotocols and does not use Sun’s URLConnection classes.This class is geared toward interpreting text that will be displayed in a browser. Consequently,it has many methods for dealing with header fields and content types.You do not create a URLConnection object yourself; it is created and returned by a URL object.After you have an instance of a URLConnection, you can examine the various header fields withthe getHeaderField methods:

public String getHeaderField(String fieldName)returns the value of the header field named by fieldName. If this field is not present in the resource, this method returns null.

public String getHeaderField(int n)

returns the value of the nth field in the resource. If there are not that many header fields, this method returns null. You can get the corresponding field name with the getHeaderFieldKey method.

public int getHeaderFieldKey(int n)

returns the field name of the nth field in the resource. If there are not that many header fields, this method returns null.

You can also get a header field value as an integer or a date using the following methods:

public int getHeaderFieldInt(String fieldName, int defaultValue)
converts the header field named by fieldName to an integer. If the field does not exist or is not a valid integer, it returns defaultValue.

public int getHeaderFieldDate(String fieldName, long defaultValue)

interprets the header field value as a date and returns the number of milliseconds since the epoch for that date. If the field does not exist or is not a valid date, it returns defaultValue.

In addition to interpreting the header fields, the URLConnection class also returns information

about the content:

public String getContentEncoding()

public int getContentLength()

public String getContentType()

As with the URL class, you can get the entire content of the URL as an object using the getContent method:

public Object getContent()

throws IOException, UnknownServiceException

This method probably won't work under Netscape, but should work under HotJava.Sometimes a program tries to access a URL that requires user authentication in the form of adialog box, which automatically pops up when you open the URL. Because you do not alwayswant your Java program to require that a user be present, you can tell the URLConnection classwhether it should allow user interaction. If a situation occurs that requires user interaction andyou have turned it off, the URLConnection class will throw an exception.The setAllowUserInteraction method, when passed a value of true, will permit interactionwith a user when needed:

public void setAllowUserInteraction(boolean allowInteraction)

public boolean getAllowUserInteraction()

returns true if this class will interact with a user when needed.

public static void setDefaultAllowUserInteraction(boolean default)

changes the default setting for allowing user interaction on all new instances of URLConnection.

Changing the default setting does not affect instances that have already been created.

public static boolean getDefaultAllowUserInteraction()

returns the default setting for allowing user interaction.

Some URLs allow two-way communication. You can tell a URLConnection whether it should allow input or output by using the doInput and doOutput methods:

public void setDoInput(boolean doInput)

public void setDoOutput(boolean doOutput)

You can set either or both of these values to true. The doInput flag is true by default, and the doOutput flag is false by default.

You can query the doInput and doOutput flags with getDoInput and getDoOutput:

public boolean getDoInput()

public boolean getDoOutput()

The getInputStream and getOutputStream methods return input and output streams for the

resource:

public InputStream getInputStream()

throws IOException, UnknownServiceException

public OutputStream getOutputStream()

throws IOException, UnknownServiceException

8.13 THE *HTTPURLCONNECTION*CLASS

The HTTP protocol has some extra features that the URLConnection class does not address.

When you send an HTTP request, for instance, you can make several different requests (GET,

POST, PUT, and so on). The HTTPURLConnection class provides better access to HTTP-specificoptions.

One of the most important fields in the HTTPURLConnection is the request method. You can set the request method by calling setRequestMethod with the name of the method you want:

public void setRequestMethod(String method) throws ProtocolException

The valid methods are: GET, POST, HEAD, PUT, DELETE, OPTIONS, and TRACE. If you don’t set a requestmethod, the default method is GET. Calling getRequestMethod will return the currentmethod:

public String getRequestMethod()

When you send an HTTP request, the HTTP server responds with a response code and message.

If you try to access a Web page that no longer exists, for example, you get a “404 NotFound” message. The getResponseMessage method returns the message part of a responsewhile the getResponseCode returns the numeric portion

:

public String getResponseMessage() throws IOException

public int getResponseCode() throws IOException

In the case of “404 Not Found”, getResponseCode would return 404, and getResponseMessage would return “Not Found”.

Because Web sites move around frequently, Web servers support the notion of redirection,where you are automatically sent to a page’s new location. The HTTPURLConnection class enablesyou to choose whether it should automatically follow a redirection. Passing a flag valueof true to setFollowRedirects method instructs the HTTPURLConnection class to follow aredirection:

public static void setFollowRedirects(boolean flag)

The getFollowRedirects method returns true if redirection is turned on:

public static boolean getFollowRedirects()

The getProxy method returns true if all HTTP requests are going through a proxy:

public abstract boolean usingProxy()

New with JDK 1.2 you can also obtain the message stream that results after an HTTP error.

The getErrorStream() method returns an InputStream that will contain the data sent after an error. For instance, if the server responded with a 404, the HTTPURLConnection would throw aFileNotFoundException. However, the Web server might have sent a help page along with the 404. The getErrorStream() method would provide you with a handle to that help page.public InputStream getErrorStream()

8.14 THE *URLENCODER* CLASS

This class contains only one static method that converts a string into URL-encoded form. The URL encoding reduces a string to a limited set of characters. Only letters, digits, and the underscore character are left untouched. Spaces are converted to a +, and all other characters areconverted to hexadecimal and written as %xx, where xx is the hex representation of the character.

The format for the encode method is as follows:

public static String encode(String s)

8.15 The *URLDecode*r Class

JDK 1.2 has added a class to allow you to decode strings that are in MIME format (the strings that URLEncoder produces, for instance). When the decode process completes, all ASCII charactersa through z, A through Z, and 0 through 9 remain the same, but plus signs (+) are convertedinto spaces. The rest of the characters in the string are changed to three-character strings. These three-character strings begin with a percent sign (%) and are followed by thetwo-digit hexadecimal representation of the lower 8 bits of the character. For instance, thestring “Hello+%56” is converted to “Hello V”.

public static String decode(String s)

8.16 THE *URLSTREAMHANDLER* CLASS

The URLStreamHandler class is responsible for parsing a URL and creating a URLConnectionobject to access that URL. When you open a connection for a URL, it scans a set of packages fora handler for that URL’s protocol. The handler should be named <protocol>.Handler. If youopen an HTTP URL, for instance, the URL class searches for a class named <some packagename>.http.Handler. By default, the class only searches the package sun.net.www.protocol,but you may specify an alternate search path by setting the system property tojava.protocol.handler.pkgs. This property should contain a list of alternate packages tosearch that are separated by vertical bars. For example:mypackages.urls|thirdparty.lib|funstuff”

At the minimum, any subclass of the URLStreamHandler must implement an openConnection method:
protected abstract URLConnection openConnection(URL u) throws IOException

This method returns an instance of URLConnection that knows how to speak the correctprotocol.If you create your own URLStreamHandler for the FTP protocol, for example, this methodshould return a URLConnection that speaks the FTP protocol.You can also change the way a URL string is parsed by creating your own parseURL and setURL methods:

protected void parseURL(URL u, String spec, int start, int limit)

This method parses a URL string, starting at position start in the string and going up to position limit. It modifies the URL directly, after it has parsed the string, using the protected set method in the URL.

You can set the different parts of a URL's information using the setURL method:

protected void setURL(URL u, String protocol, String host, int port, String file, String ref)

The call to set looks like the following:

u.set(protocol, host, port, file, ref);

8.17 THE *CONTENTHANDLER* CLASS

When you fetch a document using the HTTP protocol, the Web server sends you a series ofheaders before sending the actual data. One of the items in this header indicates what kind ofdata is being sent. This data is referred to as content, and the type of the data (referred to asthe MIME content-type) is specified by the Content-type header. Web browsers use the contenttype to determine what to do with the incoming data.

If you want to provide your own handler for a particular MIME content-type, you can create a ContentHandler class to parse it and return an object representing the contents. The mechanism for setting up your own content handler is almost identical to that of setting up your ownURLStreamHandler. You must give it a name of the form <some package name>.major.minor.

The major and minor names come from the MIME Content-type header, which is in the following form:
Content-type: major/minor

One of the most common major/minor combinations is text/plain. If you define your own text/plain handler, it can be named MyPackage.text.plain. By default, the URLConnection class searches for content handlers only in a package named sun.net.www.content. You can give additional package names by specifying a list of packages separated by vertical bars in the java.content.handler.pkgs system property.

The only method you must implement in your ContentHandler is the getContent method:
public abstract Object getContent(URLConnection urlConn) throws IOException

It is completely up to you how you actually parse the content and select the kind of object you return.

8.18 THE SOCKET CLASS

The Socket class is one of the fundamental building blocks for network-based Java applications. It implements a two-way connection-oriented communications channel between programs. After a socket connection is established, you can get input and output streams from the Socket object. To establish a socket connection, a program must be listening for connections on a specific port number. Although socket communications are peer-to-peer—that is, neither end of the socket connection is considered the master, and data can be sent either way at any time—the connection establishment phase has a notion of a server and a client.

Think of a socket connection as a phone call. After the call is made, either party can talk at any time, but when the call is first made, someone must make the call and someone else must listen for the phone to ring. The person making the call is the client, and the person listening for the call is the server.
The ServerSocket class, discussed later in this chapter, listens for incoming calls. The Socket class initiates a call. The network equivalent of a telephone number is a host address and port.

The host address can either be a host name such as netcom.com, or a numeric address such as 192.100.81.100. The port number is a 16-bit number that is usually determined by the server. When you create a Socket object, you pass the constructor the destination host name and port number for the server you are connecting to. For example,

public Socket(String host, int port) throws UnknownHostException, IOExceptioncreates a socket connection to port number port at the host named by host. If the Socket class cannot determine the numeric address for the host name, it throws an UnknownHostException.

If there is a problem creating the connection—for instance, if there is no server listening at that port number—you get an IOException.creates a socket connection to port number port at the host named by host. You can optionally request this connection be made by using datagram-based communication rather than streambased.

With a stream, you are assured that all the data sent over the connection will arrivecorrectly. Datagrams are not guaranteed, however, so it is possible that messages can be lost. The tradeoff here is that the datagrams are much faster than the streams. Therefore, if you have a reliable network, you may be better off with a datagram connection. The default mode for Socket objects is Stream mode. If you pass false for the stream parameter, the connection will be made in Datagram mode. You cannot change modes after the Socket object has been created.public Socket(InetAddress address, int port) throws IOExceptioncreates a socket connection to port number port at the host whose address is stored inaddress.

public Socket(InetAddress address, int port, boolean stream) throws IOExceptioncreates a socket connection to port number port at the host whose address is stored in address.
If the stream parameter is false, the connection is made in Datagram mode.

Sending and Receiving Socket Data The Socket class does not contain explicit methods for sending and receiving data. Instead, it provides methods that return input and output streams, enabling you to take full advantage of the existing classes in java.io.

The getInputStream method returns an InputStream for the socket;

the getOutputStream method returns an OutputStream:

public InputStream getInputStream() throws IOException

public OutputStream getOutputStream() throws IOException

Getting Socket Information

You can get information about the socket connection such as the address, the port it is connected to, and its local port number.

The getInetAddress and getPort methods return the host address and port number for the other end of the connection:

public InetAddress getInetAddress()

public int getPort()

You can get the local port number of your socket connection from the getLocalPort method:

public int getLocalPort() Setting Socket Options

Certain socket options modify the behavior of sockets. They are not often used, but it is nice to have them available. The setSoLinger method sets the amount of time that a socket will spend trying to send data after it has been closed:
public void setSoLinger(boolean on, int maxTime) throws SocketException

Normally, when you are sending data over a socket and you close the socket, anyuntransmitted data is flushed. By turning on the Linger option, you can make sure that all data has been sent before the socket connection is taken down. You can query the linger time with getSoLinger:

public int getSoLinger() throws SocketException

If the Linger option is off, getSoLinger returns –1.
If you try to read data from a socket and there is no data available, the read method normally blocks (it waits until there is data). You can use the setSoTimeout method to set the maximum amount of time that the read method will wait before giving up:

public synchronized void setSoTimeout(int timeout) throws SocketException

A timeout of 0 indicated that the read method should wait forever (the default behavior). If the read times out, rather than just returning, it will throw java.io.InterruptedIOException, but the socket will remain open. You can query the current timeout with getSoTimeout:

public synchronized int getSoTimeout() throws SocketExceptionThe TCP protocol used by socket connections is reasonably efficient in network utilization. If it is sending large amounts of data, it usually packages the data into larger packets. The reason this is more efficient is that there is a certain fixed amount of overhead per network packet. If the packets are larger, the percentage of network bandwidth consumed by the overhead is much smaller. Unfortunately, TCP can also cause delays when you are sending many small packets in a short amount of time. If you are sending mouse coordinates over the network, for instance, the TCP driver will frequently group the coordinates into larger packets while it is waiting for acknowledgment that the previous packets were received. This makes the mousemovement look pretty choppy. You can ask the socket to send information as soon as possible by passing true to setTcpNoDelay:

public void setTcpNoDelay(boolean on)

The getTcpNoDelay method returns true if the socket is operating under the No Delay option (if the socket sends things immediately):

public boolean getTcpNoDelay()

Closing the Socket Connection

The socket equivalent of “hanging up the phone” is closing down the connection, which is performed by the close method:

public synchronized void close() throws IOException

Waiting for Incoming Data Reading data from a socket is not quite like reading data from a file, even though both are input streams. When you read a file, all the data is already in the file. But with a socket connection, you may try to read before the program on the other end of the connection has sent something.

Because the read methods in the different input streams all block—that is, they wait for data if none is present—you must be careful that your program does not completely halt while waiting. The typical solution for this situation is to spawn a thread to read data from the socket.

Listing 8.7 shows a thread that is dedicated to reading data from an input stream. It notifies your program of new data by calling a dataReady method with the incoming data.

Listing 8.7 Source Code for ReadThread.java

```
import java.net.*;
import java.lang.*;
import java.io.*;
/**
* A thread dedicated to reading data from a socket connection.
*/
public class ReadThread extends Thread
{
protected Socket connectionSocket; // the socket you are reading from
protected DataInputStream inStream; // the input stream from the socket
protected ReadCallback readCallback;
/**
* Creates an instance of a ReadThread on a Socket and identifies the callback
* that will receive all data from the socket.
** @param callback the object to be notified when data is ready
* @param connSock the socket this ReadThread will read data from
* @exception IOException if there is an error getting an input stream
* for the socket
*/
public ReadThread(ReadCallback callback, Socket connSock)
throws IOException
{
connectionSocket = connSock;
readCallback = callback;
inStream = new DataInputStream(connSock.getInputStream());
}
/**
* Closes down the socket connection using the socket’s close method
*/
protected void closeConnection()
{
try {
connectionSocket.close();
} catch (Exception oops) {
}
stop();
}
/**
* Continuously reads a string from the socket and then calls dataReady in the
* read callback. If you want to read something other than a string, change
* this method and the dataReady callback to handle the appropriate data.
*/
public void run()
{
while (true)
{
try {
// readUTF reads in a string
String str = inStream.readUTF();
// Notify the callback that you have a string
readCallback.dataReady(str);
}
catch (Exception oops)
{
// Tell the callback there was an error
readCallback.dataReady(null);
}
}
}
}
```

Listing 8.8 shows the ReadCallback interface, which must be implemented by a class to receive data from a ReadThread object.

The Socket Class

Listing 8.8 Source Code for ReadCallback.java

```
/**
* Implements a callback interface for the ReadConn class
*/
public interface ReadCallback
{
/**
* Called when there is data ready on a ReadConn connection.
* @param str the string read by the read thread, If null, the
* connection closed or there was an error reading data
*/
public void dataReady(String str);
}
```

A Simple Socket Client

Using these two classes, you can implement a simple client that connects to a server and uses a read thread to read the data returned by the server. The corresponding server for this client is presented in the following section, “The ServerSocket Class.” Listing 8.9 shows the SimpleClient class.

Listing 8.9 Source Code for SimpleClient.java

```
import java.io.*;
import java.net.*;
/**
* This class sets up a Socket connection to a server, spawns
* a ReadThread object to read data coming back from the server,
* and starts a thread that sends a string to the server every
* 2 seconds.
*/
public class SimpleClient extends Object implements Runnable, ReadCallback
{
protected Socket serverSock;
protected DataOutputStream outStream;
protected Thread clientThread;
protected ReadThread reader;
public SimpleClient(String hostName, int portNumber)
throws IOException
{
Socket serverSock = new Socket(hostName, portNumber);
// The DataOutputStream has methods for sending different data types
// in a machine-independent format. It is very useful for sending data
// over a socket connection.
outStream = new DataOutputStream(serverSock.getOutputStream());
// Create a reader thread
reader = new ReadThread(this, serverSock);
// Start the reader thread
reader.start();
}
// These are generic start and stop methods for a Runnable
public void start()
```

```
{
clientThread = new Thread(this);
clientThread.start();
}
public void stop()
{
clientThread.stop();
clientThread = null;
}
// sendString sends a string to the server using writeUTF
public synchronized void sendString(String str)
throws IOException
{
System.out.println("Sending string: "+str);
outStream.writeUTF(str);
}
// The run method for this object just sends a string to the server
// and sleeps for 2 seconds before sending another string
public void run()
{
while (true)
{
try {
sendString("Hello There!");
Thread.sleep(2000);
} catch (Exception oops) {
// If there was an error, print info and disconnect
oops.printStackTrace();
disconnect();
stop();
}
}
}
// The disconnect method closes down the connection to the
server
public void disconnect()
{
try {
reader.closeConnection();
} catch (Exception badClose) {
// should be able to ignore
}
}
// dataReady is the callback from the read thread. It is called
// whenever a string is received from the server.
public synchronized void dataReady(String str)
{
System.out.println(“Got incoming string: “+str);
}
public static void main(String[] args)
{
try {
/* Change localhost to the host you are running the server on. If
it
is on the same machine, you can leave it as localhost. */
SimpleClient client = new SimpleClient(“localhost”,4331);
client.start();
} catch (Exception cantStart) {
System.out.println(“Got error”);
cantStart.printStackTrace();
}
}
}
```

8.19 THE SERVERSOCKET CLASS

The ServerSocket class listens for incoming connections and creates a Socket object for eachnew connection. You create a server socket by giving it a port number to listen on:

```
public ServerSocket(int portNumber) throws IOException
```

If you do not care what port number you are using, you can have the system assign the portnumber for you by passing in a port number of 0.Many socket implementations have a notion of connection backlog. That is, if many clientsconnect to a server at once, the number of connections that have yet to be accepted are thebacklog. After a server hits the limit of backlogged connections, the server refuses any newclients. To create a ServerSocket with a specific limit of backlogged connections, pass the portnumber and backlog limit to the Constructor:

```
public ServerSocket(int portNumber, int backlogLimit)
throws IOException
```

Accepting Incoming Socket Connections

After the server socket is created, the accept method will return a Socket object for each new connection:

```
public Socket accept() throws IOException
```

If no connections are pending, the accept method will block until there is a connection. If you do not want your program to block completely while you are waiting for connections, you should perform the accept in a separate thread.When you no longer want to accept connections, close down the ServerSocket object with theclose method:

```
public void close() throws IOException
```

The close method does not affect the existing socket connections that were made through this ServerSocket. If you want the existing connections to close, you must close each one explicitly.

Getting the Server Socket Address

If you need to find the address and port number for your server socket, you can use the getInetAddress and getLocalPort methods:

```
public InetAddress getInetAddress()
public int getLocalPort()
```

The getLocalPort method is especially useful if you had the system assign the port number.

You may wonder what use it is for the system to assign the port number because you somehow must tell the clients what port number to use. There are some practical uses for this method, however. One use is implementing the FTP protocol. If you have ever watched an FTP sessionin action, you will notice that when you get or put a file, a message such as PORT command accepted appears. What has happened is that your local FTP program created the equivalent of a server socket and sent the port number to the FTP server. The FTP server then creates a connection back to your FTP program using this port number.

Writing a Server Program

You can use many models when writing a server program. You can make one big server object that accepts new clients and contains all the necessary methods for communicating with them, for example. You can make your server more modular by creating special objects that communicate with clients but invoke methods on the main server object. Using this model, you can
have clients who all share the server’s information but can communicate using different
protocols.
Listing 8.10 shows an example client handler object that talks to an individual client and passes

the client’s request up to the main server.

Listing 8.10 Source Code for ServerConn.java

```
import java.io.*;
import java.net.*;
/**
* This class represents a server’s client. It handles all the
* communications with the client. When the server gets a new
* connection, it creates one of these objects, passing it the
* Socket object of the new client. When the client’s connection
* closes, this object goes away quietly. The server doesn’t
actually
* have a reference to this object.
*
* Just for example’s sake, when you write a server using a setup
* like this, you will probably have methods in the server that
* this object will call. This object keeps a reference to the server
* and calls a method in the server to process the strings read
from
* the client and returns a string to send back.
*/
public class ServerConn extends Object implements
ReadCallback
{
protected SimpleServer server;
protected Socket clientSock;
protected ReadThread reader;
protected DataOutputStream outStream;
public ServerConn(SimpleServer server, Socket clientSock)
throws IOException
{
this.server = server;
this.clientSock = clientSock;
outStream = new
DataOutputStream(clientSock.getOutputStream());
reader = new ReadThread(this, clientSock);
reader.start();
}
/**
* This method received the string read from the client, calls
* a method in the server to process the string, and sends back
* the string returned by the server.
*/
public synchronized void dataReady(String str)
{
```

```
if (str == null)
{
disconnect();
return;
}
try {
outStream.writeUTF(server.processString(str));
} catch (Exception writeError) {
writeError.printStackTrace();
disconnect();
return;
}}
/**
* This method closes the connection to the client. If there is an error
* closing the socket, it stops the read thread, which should eventually
* cause the socket to get cleaned up.
**/
public synchronized void disconnect()
{
try {
reader.closeConnection();
} catch (Exception cantclose) {
reader.stop();
}
}
}
```

With the ServerConn object handling the burden of communicating with the clients, your server object can concentrate on implementing whatever services it should provide. Listing 8.11 shows a simple server that takes a string and sends back the reverse of the string.

Listing 8.11 Source Code for SimpleServer.java

```
import java.io.*;
import java.net.*;
/**
* This class implements a simple server that accepts incoming
* socket connections and creates a ServerConn instance to handle
* each connection. It also provides a processString method that
* takes a string and returns the reverse of it. This method is
* invoked by the ServerConn instances when they receive a string
* from a client.
```

```
*/
public class SimpleServer extends Object
{
protected ServerSocket listenSock;
public SimpleServer(int listenPort)
throws IOException
{
// Listen for connections on port listenPort
listenSock = new ServerSocket(listenPort);
}
public void waitForClients()
{
while (true)
{
try {
// Wait for the next incoming socket connection
Socket newClient = listenSock.accept();
// Create a ServerConn to handle this new connection
ServerConn newConn = new ServerConn(
this, newClient);
} catch (Exception badAccept) {
badAccept.printStackTrace();
// print an error, but keep going
}
}
}
// This method takes a string and returns the reverse of it
public synchronized String processString(String inStr)
{
StringBuffer newBuffer = new StringBuffer();
int len = inStr.length();
// Start at the end of the string and move down towards the
beginning
for (int i=len-1; i >= 0; i--) {
// Add the next character to the end of the string buffer
// Since you started at the end of the string, the first character
// in the buffer will be the last character in the string
newBuffer.append(inStr.charAt(i));
}
return newBuffer.toString();
}
public static void main(String[] args)
{
try {
```

```
// Crank up the server and wait for connection
SimpleServer server = new SimpleServer(4321);
server.waitForClients();
} catch (Exception oops) {
// If there was an error starting the server, say so!
System.out.println(“Got error:”);
oops.printStackTrace();
}
}
}
```

8.20 THE INETADDRESS CLASS

The InetAddress class contains an Internet host address. Internet hosts are identified one oftwo ways:

- Name
- Address

The address is a 4-byte number usually written in the form a.b.c.d, like 192.100.81.100. Whendata is sent between computers, the network protocols use this numeric address for determiningwhere to send the data. Host names are created for convenience. They keep you fromhaving to memorize a lot of 12-digit network addresses. It is far easier to remembernetcom.com, for example, than it is to remember 192.100.81.100.

As it turns out, relating a name to an address is a science in itself. When you make a connectionto netcom.com, your system needs to find out the numeric address for netcom. It willusually use a service called Domain Name Service, or DNS. DNS is the telephone book servicefor Internet addresses. Host names and addresses on the Internet are grouped into domainsand subdomains, and each subdomain may have its own DNS—that is, its own local phonebook.

You may have noticed that Internet host names are usually a number of names thatare separated by periods. These separate names represent the domain a host belongsto. netcom5.netcom.com, for example, is the host name for a machine named netcom5 inthe netcom.com domain. The netcom.com domain is a subdomain of the .com domain. Anetcom.edu domain could be completely separate from the netcom.com domain, andnetcom5.netcom.edu would be a totally different host. Again, this is not too different fromphone numbers. The phone number 404-555-1017 has an area code of 404, for example, whichcould be considered the Atlanta domain. The exchange 555 is a subdomain of the Atlanta domain,and 1017 is a specific number in the 555 domain, which is part of the

Atlanta domain. Justas you can have a netcom5.netcom.edu that is different from netcom5.netcom.com, you canhave an identical phone number in a different area code, such as 212-555-1017.The important point to remember here is that host names are only unique within a particulardomain. Don’t think that your organization is the only one in the world to have named its machinesafter The Three Stooges, Star Trek characters, or characters from various comic strips.

Converting a Name to an Address

The InetAddress class handles all the intricacies of name lookup for you. The getByNamemethod takes a host name and returns an instance of InetAddress that contains the networkaddress of the host:

public static synchronized InetAddress getByName(String host)throws UnknownHostExceptionA host can have multiple network addresses. Suppose, for example, that you have your ownLAN at home as well as a Point-to-Point Protocol connection to the Internet. The machine with the PPP connection has two network addresses: the PPP address and the local LAN address.

You can find out all the available network addresses for a particular host by callinggetAllByName:public static synchronized InetAddress[] getAllByName(String host)throws UnknownHostException

The getLocalHost method returns the address of the local host:

public static InetAddress getLocalHost()throws UnknownHostException

Examining the InetAddress

The InetAddress class has two methods for retrieving the address that it stores. ThegetHostName method returns the name of the host, and getAddress returns the numeric address of the host:

public String getHostName()

public byte[] getAddress()

The getAddress method returns the address as an array of bytes. Under the current Internet addressing scheme, an array of four bytes would be returned. If and when the Internet goes to a larger address size, however, this method just returns a larger array. The following codefragment prints out a numeric address using the dot notation:

```
byte[] addr = someInetAddress.getAddress();
System.out.println((addr[0]&0xff)+”.”+(addr[1]&0xff)+”.”+
(addr[2]&0xff)+”.”+(addr[3]&0xff));
```

You may be wondering why the address values are ANDed with the hex value ff (255 in decimal).

The reason is that byte values in Java are signed 8-bit numbers. That means when theleftmost bit is 1, the number is negative. Internet addresses are not usually written with negative numbers. By ANDing the values with 255, you do not change the value, but you suddenly treat the value as a 32-bit integer value whose leftmost bit is 0, and whose rightmost 8 bits represent the address.

Getting an Applet’s Originating Address

Under many Java-aware browsers, socket connections are restricted to the server where the applet originated. In other words, the only host your applet can connect to is the one it was loaded from. You can create an instance of an InetAddress corresponding to the applet’s originating host by getting the applet’s document base or code base URL and then getting the URL’s host name. The following code fragment illustrates this method:

```
URL appletSource = getDocumentBase(); // must be called from
applet
InetAddress appletAddress = InetAddress.getByName(
appletSource.getHost());
```

8.21 THE DATAGRAMSOCKET CLASS

The DatagramSocket class implements a special kind of socket that is made specifically for sending datagrams. A datagram is somewhat like a letter in that it is sent from one point to another and can occasionally get lost. Of course, Internet datagrams are several orders of magnitude faster than the postal system. A datagram socket is like a mailbox. You receive all your datagrams from your datagram socket. Unlike the stream-based sockets you read about earlier, you do not need a new datagram socket for every program you must communicate with.

If the datagram socket is the network equivalent of a mailbox, the datagram packet is the equivalent of a letter. When you want to send a datagram to another program, you create a DatagramPacket object that contains the host address and port number of the receiving DatagramSocket, just like you must put an address on a letter when you mail it. You then call the send method in your DatagramSocket, and it sends your datagram packet off through the ethernet network to the recipient.

Not surprisingly, working with datagrams involves some of the same problems as mailing letters. Datagrams can get lost and delivered out of sequence. If you write two letters to someone, you have no guarantee which letter the person will receive first. If one letter refers to the other, it could cause confusion. There is no easy solution for this situation, except to plan for the possibility.

Another situation occurs when a datagram gets lost. Imagine that you have mailed off your house payment, and a week later the bank tells you it hasn't received it. You don't know what happened to the payment—maybe the mail is very slow, or maybe the payment was lost. If you mail off another payment, maybe the bank will end up with two checks from you. If you don't mail it off and the payment really is lost, the bank will be very angry. This, too, can happen with datagrams. You may send a datagram, not hear any reply, and assume it was lost. If you send another one, the server on the other end may get two requests and become confused. A good way to minimize the impact of this kind of situation is to design your applications so that multiple datagrams of the same information do not cause confusion. The specifics of this design are beyond the scope of this book. You should consult a good book on network programming.

You can create a datagram socket with or without a specific port number:

public DatagramSocket() throws SocketException

public DatagramSocket(int portNumber) throws SocketException
As with the Socket class, if you do not give a port number, one will be assigned automatically.

You only need to use a specific port number when other programs need to send unsolicited datagrams to you. Whenever you send a datagram, it has a return address on it, just like a letter. If you send a datagram to another program, it can always generate a reply to you without you explicitly telling it what port you are on. In general, only your server program needs to have a specific port number. The clients who send datagrams to the server and receive replies from it can have system-assigned port numbers because the server can see the return addresson their datagrams.

The mechanism for sending and receiving datagrams is about as easy as mailing a letter and checking your mailbox—most of the work is in writing and reading the letter. The send method sends a datagram to its destination (the destination is stored in the DatagramPacket object):public void send(DatagramPacket packet) throws IOException.

The receive method reads in a datagram and stores it in a DatagramPacket object:public synchronized void receive(DatagramPacket packet) throws IOException

When you no longer need the datagram socket, you can close it down with the close method:

public synchronized void close()

Finally, if you need to know the port number of your datagram socket, the getLocalPort

method gives it to you:

public int getLocalPort()

8.22 THE DATAGRAMPACKET CLASS

The DatagramPacket class is the network equivalent of a letter. It contains an address and other information. When you create a datagram, you must give it an array to contain the data as well as the length of the data. The DatagramPacket class is used in two ways:

- *As a piece of data to be sent out over a datagram socket*. In this case, the array used to create the packet should contain the data you want to send, and the length should be the exact number of bytes you want to send.
- *As a holding place for incoming datagrams*. In this case, the array should be large enough to hold whatever data you are expecting, and the length should be the maximum number of bytes you want to receive.

To create a datagram packet that is to be sent, you must give not only the array of data and the length, but you must also supply the destination host and port number for the packet:public DatagramPacket(byte[] buffer, int length, InetAddress destAddress,int destPortNumber)

When you create a datagram packet for receiving data, you only need to supply an array large enough to hold the incoming data, as well as the maximum number of bytes you wish to receive:public DatagramPacket(byte[] buffer, int length)
The DatagramPacket class also provides methods to query the four components of the packet:

public InetAddress getAddress()

For an incoming datagram packet, getAddress returns the address that the datagram wassent from. For an outgoing packet, getAddress returns the address where the datagram willbe sent.

public int getPort()

For an incoming datagram packet, this is the port number that the datagram was sent from.

For an outgoing packet, this is the port number where the datagram will be sent.

public byte[] getData()

public int getLength()

Broadcasting Datagrams:

A datagram broadcast is the datagram equivalent of junk mail. It causes a packet to be sent to a number of hosts at the same time. When you broadcast, you always broadcast to a specific port number, but the network address you broadcast to is a special address.Recall that Internet addresses are in the form a.b.c.d. Portions of this address are consideredyour host address, and other portions are considered your network address. The networkaddress is the left portion of the address; the host address is the right portion. The dividingline between them varies based on the first byte of the address (the a portion). If a is less than128, the network address is just the a portion, and the b.c.d is your host address. This addressis referred to as a Class A address. If a is greater than or equal to 128 and less than 192, thenetwork address is a.b, and the host address is c.d. This address is referred to as a Class Baddress. If a is greater than or equal to 192, the network address is a.b.c, and the host addressis d. This address is referred to as a Class C address.Why is the network address important? If you want to be polite, you should only broadcast toyour local network. Broadcasting to the entire world is rather rude and probably won’t workanyway because many routers block broadcasts past the local network. To send a broadcast toyour local network, use the numeric address of the network and put in 255 for the portions thatrepresent the host address. If you are connected to Netcom, for example, which has a networkaddress that starts with 192, you should only broadcast Netcom’s network of 192.100.81, whichmeans the destination address for your datagrams should be 192.100.81.255. On the otherhand, you might be on a network such as 159.165, which is a Class B address. On that network,you would broadcast to 159.165.255.255. You should consult your local system administratorabout this, however, because many Class A and Class B networks are locally subdivided. Youare safest just broadcasting to a.b.c.255 if you must broadcast at all.

A Simple Datagram Server Listing 8.12 shows a simple datagram server program that just echoes back any datagrams itreceives.

Listing 8.12 Source Code for DatagramServer.java

```
import java.net.*;
/**
* This is a simple datagram echo server that receives datagrams
* and echoes them back untouched.
*/
public class DatagramServer extends Object
{
```

```
public static void main(String[] args)
{
try {
// Create the datagram socket with a specific port number
DatagramSocket mysock = new DatagramSocket(5432);
// Allow packets up to 1024 bytes long
byte[] buf = new byte[1024];
// Create the packet for receiving datagrams
DatagramPacket p = new DatagramPacket(buf,
buf.length);
while (true) {
// Read in the datagram
mysock.receive(p);
System.out.println(“Received datagram!”);
// A nice feature of datagram packets is that there is only one
// address field. The incoming address and outgoing address are
// really the same address. This means that when you receive
// a datagram, if you want to send it back to the originating
// address, you can just invoke send again.
mysock.send(p);
}
} catch (Exception e) {
e.printStackTrace();
}
}
}
```

Listing 8.13 shows a simple client that sends datagrams to the server and waits for a reply. If the datagrams get lost, however, this program will hang because it does not resend datagrams.

Listing 8.13 Source Code for DatagramClient.java

```
import java.net.*;
/**
* This program sends a datagram to the server every 2 seconds
and waits
* for a reply. If the datagram gets lost, this program will hang
since it
* has no retry logic.
*/
public class DatagramClient extends Object
{
public static void main(String[] args)
{
```

```
try {
// Create the socket for sending
DatagramSocket mysock = new DatagramSocket();
// Create the send buffer
byte[] buf = new byte[1024];
// Create a packet to send. Currently just tries to send to the
local host.
// Change the inet address to make it send somewhere else.
DatagramPacket p = new DatagramPacket(buf,
buf.length, InetAddress.getLocalHost(), 5432);
while (true) {
// Send the datagram
mysock.send(p);
System.out.println(“Client sent datagram!”);
// Wait for a reply
mysock.receive(p);
System.out.println(“Client received datagram!”);
Thread.sleep(2000);
}
} catch (Exception e) {
e.printStackTrace();
}
}
}
```

8.23 MULTICAST SOCKETS

IP multicasting is a fairly new technology that represents an improvement over simple broadcasting.A multicast functions like a broadcast in that a single message gets sent to multiple recipients, but it is only sent to recipients that are looking for it.The idea behind multicasting is that a certain set of network addresses are set aside as beingmulticast addresses. These addresses are in the range 225.0.0.0 to 239.255.255.255.Actually, network addresses between 224.0.0.0 and 224.255.255.255 are also IPmulticast addresses, but they are reserved for non-application uses. Each multicast address is considered a group. When you want to receive messages from acertain address, you join the group. You may have set up the address 225.11.22.33 as the multicast address for your stock quote system, for example. A program that wanted to receive stock quotes would have to join the 225.11.22.33 multicast group.To send or receive multicast data, you must first create a multicast socket. A multicast socket issimilar to a datagram socket (in fact, MulticastSocket is a subclass of DatagramSocket). Youcan create the multicast socket with a default port number or you can specify the port numberin the Constructor:

public MulticastSocket() throws IOException

public MulticastSocket(int portNumber) throws IOException

To join a multicast address, use the joinGroup method; to leave a group, use the leaveGroup

method:

public void joinGroup(InetAddress multicastAddr) throws IOException

public void leaveGroup(InetAddress multicastAddr) throws IOException

Multicast Sockets

NOTE:

On certain systems, you may have multiple network interfaces. This can cause a problem for multicasting because you need to listen on a specific interface. You can choose which interface your multicast socket uses by calling setInterface:

public void setInterface(InetAddress interface) throws SocketException

If your machine had IP addresses of 192.0.0.1 and 193.0.1.15, and you wanted to listen for multicast messages on the 193 network, for example, you would set your interface to the 193.0.1.15 address. Of course, you need to know the host name for that interface. You might have host names of myhost_neta for the 192 network and myhost_netb for the 193 network. In this case, you would set your interface this way:mysocket.setInterface(InetAddress.getByName(“myhost_netb”));
You can query the interface for a multicast socket by calling getInterface:public InetAddress getInterface() throws SocketException

The key to multicast broadcasting is that you must send your packets out with a “time to live” value (also called TTL). This value indicates how far the packet should go (how many networks it should jump to). A TTL value of 0 indicates that the packet should stay on the local host. A TTL value of 1 indicates that the packet should only be sent on the local network. After that, the TTL values have more nebulous meanings. A TTL value of 32 means that the packet should only be sent to networks at this site. A TTL value of 64 means the packet should remain within this region, and a value of 128 means it should remain within this continent. A value of 255 means that the packet should go everywhere. Like broadcast datagrams, it is considered rudeto send your packets to everyone. Try to limit the scope of your packets to the local network or, at least, the local site.When you send a multicast datagram, you use a special version of the send method that takes aTTL value (if you use the default send method, the TTL is always 1):public synchronized void send(DatagramPacket packet,byte timeToLive) throws IOException

You should also bear in mind that untrusted applets are not allowed to create MulticastSocket objects.

8.24 WHAT NECESSITATES JAVA SECURITY?

In any description of the features of the Java environment, a phrase such as “Java is secure” willbe found. Security can mean a lot of different things, and when you’re developing Java applets,it is critical to understand the implications of Java security. Your applets are restricted to unctioningwithin the Java security framework, which affects your design while enabling the safe execution of network-loaded code.To ensure an uncompromised environment, the Java security model errs on the side of caution.All applets loaded over the network are assumed to be potentially hostile and are treated withappropriate caution. This fact will greatly restrict your design. To enable Java applets to expandbeyond these limitations, the Java Security API has been developed.To appreciate the intent and rationale behind the framework on which Java is based, you mustinvestigate what makes security an issue at all. Java provides many solutions to matters ofsecurity, many of which will have ramifications on how you approach the installation andauthoring of Java applications in your Internet network solutions.The Internet forms a vast shared medium, allowing machines throughout the world to communicatefreely. Trusted and untrusted computers, allowing access to millions of individuals withunknown intentions, are linked together. One computer can send information to almost anyother on the Internet. Furthermore, Internet applications and protocols are not foolproof; atvarious levels, the identities can be concealed through various techniques.Adding Java to this scene opens tremendous potential for abuse. Java’s strengths present themost problematic issues, specifically these:

- Java is a full-fledged programming language that allows applications to use manyresources on the target machine, such as manipulating files, opening network sockets toremote systems, and spawning external processes.
- Java code is downloaded from the network, often from machines over which you have nocontrol. Such code could contain fatal flaws or could have been altered by a maliciousintruder. The original author could even have questionable motives that would have animpact on your system.
- Java code is smoothly, seamlessly downloaded by Java-enabled browsers such asHotJava, Netscape, or Internet

Explorer. These special programs, known as applets, canbe transferred to your machine and executed without your knowledge or permission.

This “executable content” can have capabilities that extend far beyond the originallimitations of your Web browser’s design, precisely because Java is intended to allowyour browser’s capabilities to be extended dynamically.

Given these characteristics, it is easy to see why Java code should be treated with great care.

Without a tightly controlled environment, one could envision a number of problematic scenarios:

- A malicious piece of code damages files and other resources on your computer.
- While perhaps presenting a useful application, code silently retrieves sensitive data fromyour system and transmits it to an attacker’s machine.
- When you merely visit a Web page, a virus or worm is loaded that proceeds to spreadfrom your machine to others.
- A program uses your system as a launching pad for an attack on another system, thusobscuring the identity of the real villain while perhaps misidentifying you as the truesource of the attack.
- Code created by a programmer whose abilities are not equal to yours creates a buggyprogram that unintentionally damages your system.With these problems in mind, the overall problem can be seen. To be practical, Java must providea controlled environment in which applications are executed. Avenues for abuse or unintendeddamage must be anticipated and blocked. System resources must be protected. To besafe, Java must assume code that is loaded over the network comes from an untrusted source;only those capabilities known to be secure should be permitted. However, Java should not beso restricted that its value goes unrealized.

For those who are familiar with Internet security systems, the issues Java faces are not new.

This situation presents the old paradox in which computers must have access to capabilities and resources to be useful. However, in an inverse relationship, the more power you provide to such systems, the greater the potential for abuse. In such a situation, a paranoid stance will render the system useless. A permissive stance will eventually spell doom. A prudent stance strives to find an intelligent middle ground.

8.25 THE JAVA SECURITY FRAMEWORK

The security framework provides Java with a clear API to create secure execution environments.As you know, Java consists of many different layers that create the complete Java execution environmet:

- Java language
- Feature-rich, standard API
- Java compiler
- Bytecode
- Dynamic loading and checking libraries at runtime
- Garbage collector
- Bytecode interpreter

At critical points throughout this structure, specific features help ensure a safe execution environment.
In isolation, each portion might provide little or no benefit to the system. In concert,these features work to create the solid and secure framework that makes Java a practical solution to executable content.

The Java Security Framework:

Part One: The Safety Provided by the Language:

The Java language itself provides the first layer of network security. This security provides the features that are necessary to protect data structures and limit the likelihood of unintentionally flawed programs.

Java-Enforced Adherence to the Object-Oriented Paradigm Private data structures andmethods are encapsulated within Java classes. Access to these resources is provided only through a public interface that is furnished by the class. Object-oriented code often proves to be more maintainable and follows a clear design. This helps ensure that your program does not accidentally corrupt itself, or clobber other Java elements running in the same VM.

No Pointer Arithmetic Java references cannot be incremented or reset to point to specific portions of the JVM’s memory. This means that you cannot unintentionally (or maliciously) overwrite the contents of an arbitrary object. Nor can you write data into sensitive portions of the system’s memory. Furthermore, every object that isn’t waiting for garbage collection must have a reference defined to it, so objects can’t be accidentally lost. Array-Bounds Checking Arrays in Java are bound to their known size. So in Java, if you have an array int[5] and you attempt to reference [8], your program will throw an exception (ArrayIndexOutOfBoundsException). Historically, because other languages did not pay attention to array bounds, many security problems were created. For instance, a flawed application could be induced to iterate beyond the end of an array, and when beyond the end of the array,the program would be referring to data that did not belong to the array. Java prevents this problem by ensuring that any attempt to index an element before the beginning or after the end of an array will throw an exception.

Java's Typecasting System Java ensures that any cast of one object to another is actually a legal operation. An object cannot be arbitrarily cast to another type. So, assuming that you have an object such as a Thread, if you try to cast it to an incompatible class such as a System (that is, System s = (System) new Thread()), the runtime will throw an exception (ClassCastException). In languages that do not check for cast compatibility, it is possible to cast two objects, and incorrectly manipulate objects, or thwart inheritance restrictions.

Language Support for Thread-Safe Programming Multithreaded programming is an intrinsic part of the Java language, and special semantics ensure that different threads of execution modify critical data structures in a sequential, controlled fashion.

Final Classes and Methods Many classes and methods within the Java API are declared final, preventing programs from further subclassing or overriding specific code.

Part Two: The Java Compiler

The Java compiler converts Java code to bytecode for the JVM. The compiler ensures that all the security features of the language are imposed. A trustworthy compiler establishes that the code is safe and establishes that a programmer has used many of the security features. The compiler makes sure that any information that is known at compile time is legitimate—for instance, by checking for typecasting errors.

Part Three: The Verifier

Java bytecode is the essence of what is transmitted over the network. It is machine code for the JVM. Java's security would be easy to subvert if only the policies defined previously were assumedto have been enforced. A hostile compiler could be easily written to create bytecode thatwould perform dangerous acts that the well-behaved Java compiler would prevent.Thus, security checks on the browser side are critical to maintaining a safe execution environment.

Bytecode cannot be assumed to be created from a benevolent compiler, such as javac,within the JDK. Instead, a fail-safe stance assumes that class files are hostile unless clearlyproven otherwise.To prove such an assertion, when Java bytecode is loaded, it first enters into a system knownas the verifier. The verifier performs several checks on all class files loaded into the Java executionenvironment. The verifier goes through these steps before approving any loaded code:

1. The first pass-over ensures that the class file is of the proper general format.
2. The second check ensures that various Java conventions are upheld, such as checking that every class has a superclass (except the Object class) and that final classes and methods have not been overridden.
3. The third step is the most detailed inspection of the class file. Within this step, thebytecodes themselves are examined to ensure their validity. This mechanism within the verifier is generally referred to as the bytecode verifier.
4. The last step performs some additional checks, such as ensuring the existence of class fields and the signature of methods.

For more detailed information on the verifier, read the paper by Frank Yellin titled “Low Level Security,” available at http://java.sun.com/sfaq/verifier.html. n

Part Four: The ClassLoader:

Bytecode that has reached this stage has been determined to be valid; it then enters theClassLoader, an object that subclasses the abstract class java.lang.ClassLoader. TheClassLoader loads applets incoming from the net and subjects them to the restrictions of the Applet Security Manager, described in the section titled “Part Five: Establishing a Security Policy,” later in this chapter. It strictly allocates namespaces for classes that are loaded into the runtime system. A namespace is conceptual real estate in which an object’s data structures can reside.

The ClassLoader ensures that objects don’t intrude into each other’s namespaces in unauthorized fashions. Public fields and methods can be accessed, but unless such an interface is defined, another object has no visibility to the variables. This point is important because system resources are accessed through specific classes—ones that are trusted to behave well and are installed within the JDK. If untrusted code was able to manipulate the data of the core Java API, disastrous results would ensue.

The Java Security Framework:

The ClassLoader also provides a strategic gateway for controlling which class code can beaccessed. For example, applets are prevented from overriding any of the built-in Java classes,such as those that are provided within the Java API. Imported classes are prevented from mpersonatingbuilt-in classes that are allowed to perform important system-related tasks. When a reference to an object is accessed, the namespace of built-in classes is checked first, thwarting any spoofing by network-loaded classes.

Part Five: Establishing a Security Policy:

The previous pieces of the Java security framework ensure that the Java system is not subverted by invalid code or a hostile compiler. Basically, they ensure that Java code plays by therules. Given such an assurance, you are now able to establish a higher-level security policy.This security policy

exists at the application level, allowing you to dictate what resources a Javaprogram can access and manipulate.

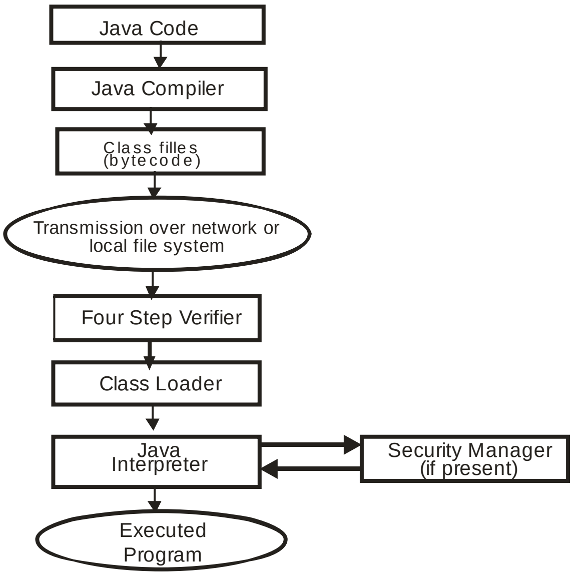
The Java API provides the java.lang.SecurityManager class as a means of creating a clearlydefined set of tasks an application can and cannot perform, such as access files or networkresources. Java applications don’t start out with a SecurityManager, meaning that all resourcesit could restrict are freely available. However, by implementing a SecurityManager, you canadd a significant measure of protection.

Java-enabled browsers use the SecurityManager to establish a security policy that greatlydistinguishes what Java applets and Java applications can do. Later in this chapter, in the section“The SecurityManager Class,” such special restrictions are described in detail.

Figure 34.1 illustrates how these separate pieces of the framework interlock to provide a safe, secure environment. This careful structure establishes an intelligent, fail-safe stance for the execution of Java programs:

- The Java language provides features that make a safe system possible.
- Such code is compiled into bytecode, where certain compile-type checks are enforced.
- Code is loaded into the Java execution environment and checked for validity by the
- verifier, which performs a multistep checking process.
- The ClassLoader ensures separate namespaces for loaded class files, allowing the Java interpreter to actually execute the program.
- The SecurityManager maintains an application-level policy, selectively permitting or denying certain actions.

Figure 8.5



The SecurityManager Class:

Most of the security features that are added to Java applets are imposed by the classjava.lang.SecurityManager, although (as previously mentioned) the use of a ClassLoader instance plays a significant role as well. The SecurityManager class allows you to establish a specific security policy that is appropriate to the level of trust given to a particular program.

This abstract class provides the capability to create an object that determines whether an operation that a program intends to perform is permitted.

The SecurityManager has methods for performing the following acts to enforce a security policy:

- Determine whether an incoming network connection from a specific host on a specific port can be accepted.
- Check whether one thread can manipulate another thread of ThreadGroup.
- Check whether a socket connection can be established with a remote system on a specific port.
- Prevent a new ClassLoader from being created.
- Prevent a new SecurityManager from being created, which could override the existing policy.
- Check whether a file can be deleted.
- Check whether a program can execute a program on the local system.
- Prevent a program from exiting the Java Virtual Machine.
- Check whether a dynamic library can be linked.
- Check whether a certain network port can be listened to for an incoming connection.

- Determine whether a program can load in specific Java packages.
- Determine whether a program can create new classes within a specific Java package.
- Identify which system properties can be accessed through the System.getProperty() method.
- Check whether a file can be read.
- Check whether data can be written to a file.
- Check whether a program can create its own implementation of network sockets.
- Establish whether a program can create a top-level window. If prevented from doing so, any windows that are allowed to be created should include some sort of visual warning.

The Security Policy of Java BrowsersWithin Web browsers, a specific policy has been identified for the loading of untrusted applets.The SecurityManager performs various checks on a program’s allowed actions; theClassLoader, which loads Java classes over the network, ensures that classes loaded fromexternal systems do not subvert this security stance. By default, the following restrictions apply:

- Applets are not allowed to read files on the local system. For example, this fails in anapplet:

  ```
  File readFile = new File(“/etc/passwd”);
  FileInputStream readIn = new FileInputStream(readFile);
  ```

- Applets are not allowed to create, modify, or delete files on the local system. For example,

  this fails in an applet:

  ```
  File writeData = new File(“write.txt”); // Can’t create files.
  FileOutputStream out = new FileOutputStream(writeData);
  out.write(1);
  File oldName = new File(“one.txt”); // Can’t modify files,such as
  File newName = new File(“two.txt”); // by changing their names
  oldName.renameTo(newName); // within directories.
  File removeFile = new File(“import.dat”); // Can’t delete files.
  removeFile.delete();
  ```

- Applets cannot check for the existence of a file on the local system. For example, this fails in an applet:

  ```
  File isHere = new File(“grades.dbm”);
  isHere.exists();
  ```

- Applets cannot create a directory on the local system. For example, this fails in an applet:

  ```
  File createDir = new File(“mydir”);
  createDir.mkdir();
  ```

- Applets cannot inspect the contents of a directory. For example, this fails in an applet:

  ```
  String[] fileNames;
  File lookAtDir = new File(“/users/hisdir”);
  fileNames = lookAtDir.list();
  ```

- Applets cannot check various file attributes, such as a file’s size, its type, or the time of the last modification. For example, this fails in an applet:

  ```
  File checkFile = new File(“this.dat”);
  long checkSize;
  boolean checkType;
  long checkModTime;
  Applet Restrictions
  checkSize = checkFile.length();
  checkType = checkFile.isFile();
  checkModTime = checkFile.lastModified();
  ```

- Applets cannot create a network connection to a machine other than the one from which the applet was loaded. This rule holds true for connections that are created through any of the various Java network classes, including java.net.Socket, java.net.URL, and java.net.DatagramSocket.

  For example, assuming that the applet was downloaded from www.untrusted.org, the following code will fail in an applet:

  ```
  // Can’t open TCP socket.
  Socket mailSocket = new Socket(“mail.untrusted.org”,25);
  // The URL objects are similarly restricted.
  URL untrustedWeb = new URL(“http://www.untrusted.org/”);
  URLConnection agent = untrustedWeb.openConnection();
  agent.connect();
  // As are UDP datagrams.
  InetAddress thatSite = new InetAddress(“www.untrusted.org”);
  int thatPort = 7;
  byte[] data = new byte[100];
  DatagramPacket sendPacket =
  new DatagramPacket(data,data.length,thatSite,thatPort);
  DatagramSocket sendSocket = new DatagramSocket();
  ```

```
sendSocket.send(sendPacket);
```

- Applets cannot act as network servers, listening for or accepting socket connections

  from remote systems. For example, this fails in an applet:

  ```
  ServerSocket listener = new ServerSocket(8000);
  listener.accept();
  ```

- Applets are prevented from executing any programs that reside on the local computer.

  For example, this fails in an applet:

  ```
  String command = “DEL \AUTOEXEC.BAT”;
  Runtime systemCommands = Runtime.getRuntime();
  systemCommands.exec(command);
  ```

- Applets are not allowed to load dynamic libraries or define native method calls. Forexample, this fails in an applet:

  ```
  Runtime systemCommands = Runtime.getRuntime();
  systemCommands.loadLibrary(“local.dll”);
  ```

- Within the Java environment, various standard system properties are set. Theseproperties can be accessed with the java.lang.System.getProperty(String key) method. Applets are allowed to read only certain system properties and are preventedfrom accessing others. Table 34.1 shows these system properties.
- Applets cannot manipulate any Java threads other than those within their own threadgroup.
- Applets cannot shut down the JVM. For example, this fails in an applet:

  ```
  // This mechanism fails.
  Runtime systemCommands = Runtime.getRuntime();
  systemCommands.exit(0);
  // As does this mechanism.
  System.exit(0);
  ```

- Applets cannot create a SecurityManager or ClassLoader instance. The Java browsercreates such an object and uses it to impose the security policy on all applets.
- The java.net package uses factories to establish particular implementations of specificconcepts: protocol handlers, content handlers, and sockets. Applets cannot override the specification of these classes: java.net.URLStreamHandlerFactory,java.net.ContentHandlerFactory, and java.net.SocketImplFactory.

As you might imagine, this policy presents some severe limitations that affect what yourapplets can and cannot do. One particular problem is that the Internet, by its very nature, is adistributed system. However, Java applets are prevented from accessing this web of computers—they can connect only to the machine from which they were downloaded.

Furthermore, because data cannot be written to the local system, applets cannot maintain apersistent state across executions on the client. As a workaround, applets must connect to aserver to store state information, reloading that information from the original server whenexecuted later.
The current Java API provides the framework for creating specialized security policies fortrusted applets loaded from known sources. This latter solution is described later in this chapter.

Java Security ProblemsDespite its success and significant attention, Java is still not a completely mature system. Sincethe release of the first version of Java, various practical flaws have been identified, and subsequentlyfixed. Understanding these flaws will provide you with a feel for the medium into whichyou are immersing yourself.An important point to note in this regard is the degree of openness that has been encouragedwithin the Java development arena. Obviously, companies such as Sun and others that have asignificant stake in promoting Java suffer when a bug or flaw is revealed. Nevertheless, publicscrutiny and critiques have been encouraged and generally well received.Based on the experience of most security professionals, such public review is an essentialcomponent of the development of a secure system. In most cases, it is impossible to prove thata system is secure. A safe stance is to assume that a system with no known flaws is merely onewith flaws that are waiting to be exposed and exploited. Peer review allows for various expertsto search for these hidden flaws—a process that is very familiar withinthe Internet community.Java's evolution has followed this philosophy, and from most practical observations, it appearsthat everyone has benefited.The opposing argument is that exposing the implementation of the system to the public allowsuntrusted and malicious individuals to identify and act on flaws before others can rectify thesituation; by keeping a system secret, it is less likely that abusive hackers will discover theseproblems. Many people experienced with Internet security disagree, believing that obscuringthe implementation is unwise: secrecy in design creates a system that is ultimately poorer, while providing more opportunity for malevolence.

8.26 THE JAVA SECURITY API: EXPANDING THE BOUNDARIESFOR APPLETS

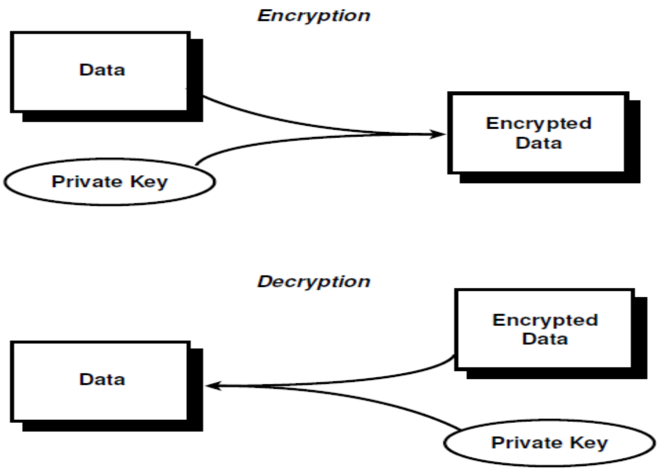
By now, you have come to realize the significant, though prudent, default limitations to whichJava applets are held. These policies create a safe but restricted environment. When you'redesigning an applet to accomplish certain tasks, you must create cumbersome workarounds,and other goals just can't be accomplished through applets.This situation is necessary because all applets are treated as hostile—a fail-safe stance. In manysituations, however, you are able to assert that certain programs are not hostile. For instance,applets distributed by a faithful vendor or provided from within your firewall can be reasonablyexpected to have greater access to system resources than a random applet loaded fromsomeone’s Web page.

One of the key capabilities missing from the initial Java implementations was the capability to establish trust relationships. Java 1.1 added the foundation of the Java Security API. Using the Security API, you have the ability to create trusted relationships with program developers and verify that code from these sources is not altered by an outside party. However, the 1.1 solution simply said that a program was either trusted or untrusted; there was no incremental controlover how trusted an applet could be. With 1.2, the security model has been extended to enablefinite trusted control.The features of the Java Security API are based on computer cryptography designs and algorithms.A quick investigation of these concepts can help you understand how the Security APIworks.

Symmetric Cryptography:

The cryptographic scheme that is most familiar to many is symmetric cryptography, or privatekeyencryption. The concept is that a special formula or process takes a piece of data and usesa special key, such as a password, to produce an encrypted block of data.The Java Security API: Expanding the Boundaries for AppletsGiven only the encrypted data, or ciphertext, it is difficult or impossible to reproduce the originalcopy. With the key, however, you can decrypt the ciphertext into the original message.Thus, anyone with access to the key can easily decrypt the data. Because the security of thissystem depends on the secrecy of this key, this scheme is referred to as private key encryption.It is symmetrical in nature because the same key that is used to encrypt the data is requiredto decrypt the message. Figure 8.6 illustrates the private key encryption scheme.

Figure 8.6

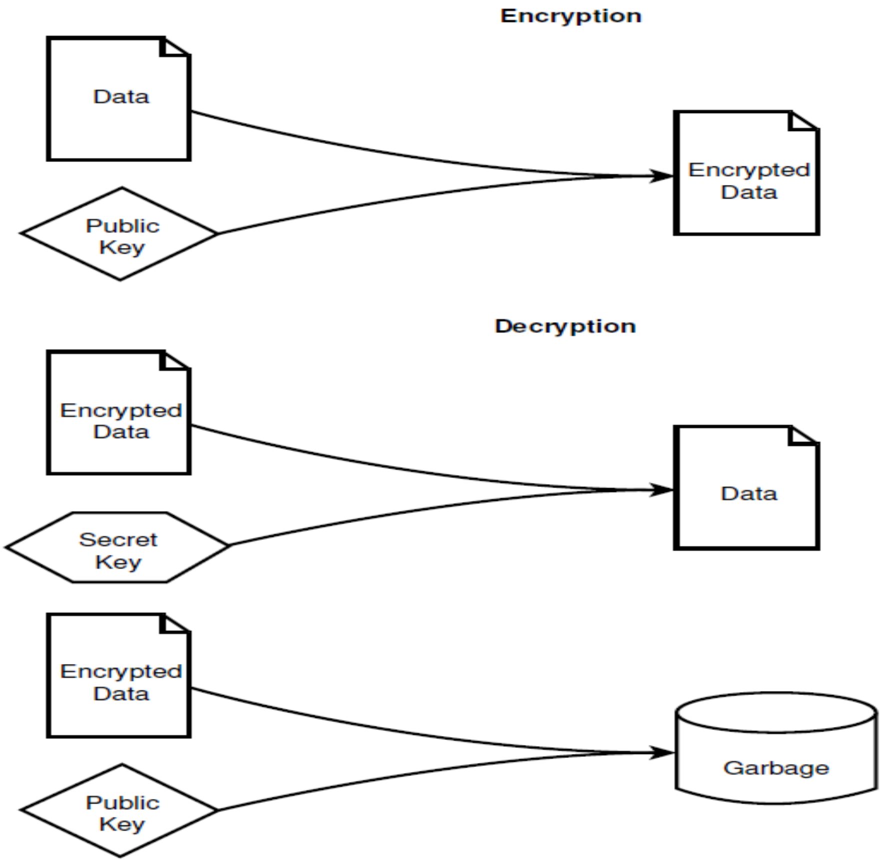


A number of cryptographic systems use private key cryptography. Data Encryption Standard(DES) is a widely used system; however, cracking it is practical with today’s technology. IDEAis a much newer algorithm and is believed to be much more secure than DES, although it hasnot been as thoroughly tested as DES. RC2 and RC4 are propriety algorithms distributed byRSA Data Security.One of the problems with using private key encryption to protect communications is that bothparties must have the same key. However, this exchange of private keys must be protected.Thus, to securely transmit documents, a secure mechanism of exchanging information mustalready exist.

Public Key Cryptography:

Public key cryptography is a phenomenal idea. It is a radical system that is based on breakthroughsmade during the 1970s. The concept is based on special mathematical algorithms.A special formula is used to create two keys that are mathematically related, but neither can beinduced from the other. One key is used to encrypt a particular message to produce aciphertext. The other key is used to decrypt the message; however, the original key cannot beused to decrypt the ciphertext. Thus, this type of cryptography is referred to as asymmetric.This system solves the problem of key distribution that limits private key cryptography. Anindividual who expects to receive protected documents can advertise one of the keys, generallyreferred to as the public key. Anyone who wants to send an encrypted message to this personmerely picks up the public key and creates the ciphertext. This encrypted message can besafely transmitted because only the other key can decrypt it. The recipient keeps the corresponding,or secret, key hidden from others because it is the only key that can be used to readmessages encrypted by the public key. Figure 8.7 shows this mechanism.

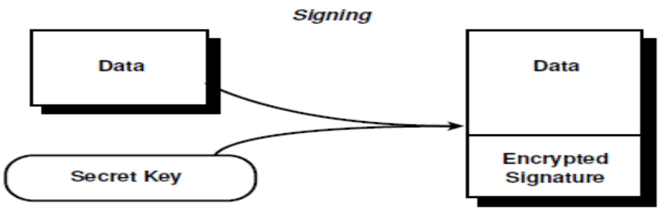
Figure 8.7

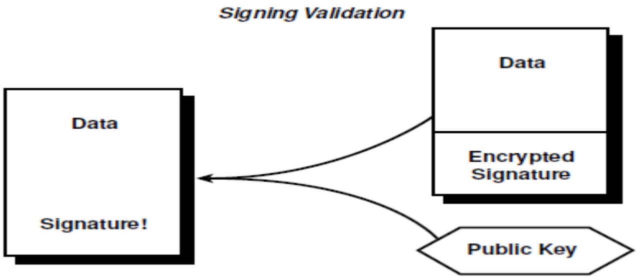


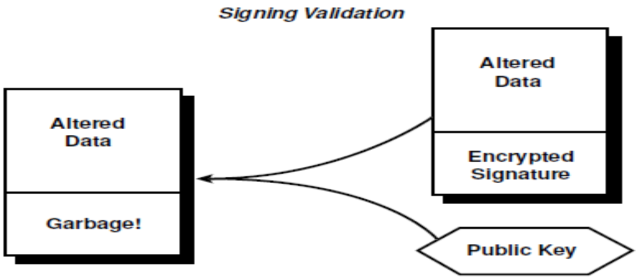
Perhaps of more usefulness to Java applets, however, is the converse operation that is known assigning. Given a message, the secret key is used to create an encrypted signature. Theunencoded message is transmitted along with the signature, and if the message is altered, thesignature cannot be decrypted. Anyone who receives the message can obtain the freely availablepublic key to ensure two things:

- The message truly was from the supposed author.
- The message was not altered in any way after being signed.

The process of signing messages through public key cryptography is shown in Figure 8.8.







Certification Authorities:

One of the limitations in the public key system is verifying that a public key truly belongs to theindividual you believe it belongs to. It is conceivable that a hostile individual could send you amessage signed with a secret key, claiming to be from another party. This attacker thenadvertisesa public key as belonging to the impersonated person. You retrieve this key and decryptthe signature. Believing that you have verified the author, you now trust information that, unbeknownstto you, is written by a hostile source.Secure transmission systems on the Web have turned to a system known as Certification Authorities(CA) to overcome this limitation. Basically, a CA is an organization or company that isvery well known and goes to great lengths to ensure that its public key is properly advertised.The CA then signs the key of other agencies that conclusively prove their identity. When youreceive the public key of this agency, you can use the CA’s public key to verify it. If successful,you know that the CA believes that this agency is what it claims to be. Thus, the CA certifiesthe agency.

If your Web browser implements a mechanism of secure communications, such as SSL, youcan see a list of some certificate authorities. Navigator is SSL enabled—if you choose Options,Security Preferences, Site Certificates, you can see the certificates of the CAs distributed withthe browser.

Key Management:

Key management is an extremely important aspect of security. You must keep your database of certificates up to date and keep your private keys secret. If you keep keys and certificates inseparate files scattered around your system, you might accidentally place a private key in apublic directory where someone could steal it. To help you with key management, Java 1.1included a key database and a key management tool called javakey. Now, with JDK 1.2, thedatabase and the capability to sign code have been split into two separate tools: keytool andjarsigner.Unfortunately, the keytool and jarsigner tools are not compatible with the javakey tool fromJDK 1.1. So if you are using 1.1 for any reason, you need to look into how the javakeyworks, and not use keytool and jarsigner. The keytool program included with JDK 1.2 is designed to allow you to create, modify, andremove keys and certificates. It stores these recordsin a new type of database called akeystore.

A certificate is a digitally signed object that is issued by a known entity, which identifies thepublic key of another entity. For instance, you could have a certificate from RSA that tells youwhat someone’s public key is. Then, when data is digitally signed, you can verify that the ignaturewas really generated by that person by checking the certificate.You can do two things when verifying a signature. First, using the certificate, you can checkthe data’s integrity. The integrity of the data means that the data has not been modified ortampered with since the time it was signed. Second, you can verify the authenticity of the data.Authenticity verifies that the person who signed the document is really who he claims to be.A certificate is generally held by an entity. An entity is a person or an organization that is ableto digitally sign information. Because signing requires a key set, a signer has both a public keyand a private key, as well as a certificate authenticating the public key.A key is a number that is associated with the entity. The public key is designed so that everyonewho needs to interact with the entity can have access to the number. A private key, on theother hand, is designed so that only the entity will know it. The two keys are mathematicallymatched so that when a value is encrypted by the public key, it can be unencrypted only by theprivate key. In addition, they are designed so that the private key cannot be derived just byknowing the public key.To store keys for an entity, you must first create an entry in the keystore database. When youcreate the entry, you must give the entity a name, and a password to access the entity. Thefollowing command creates an entry for a signer named mark:

```
keytool –genkey –alias usingjava –keypass goodbooks
  –dname “cn=QUE” –storepass zippydoda
```

The -genkey option indicates that you are creating an entry for a signer. The -aliase optionindicates that you are creating an entry for an alias that is identified directly after the option.The -keypass is used to identify the password that will be required any time you want to accessor modify the key.After you have created an entry for an entity, you can add keys and certificates for that entity.For example, suppose you received Verisign's public key in a file called vskey.key and thecertificate for that key in a file called vskey.cer. Use the -import flag on the keytool commandto import the public key into the key database:

```
keytool –import –alias verisign –file vskey.cer –keypass verisignpas
```

You can list the entities in the database with the -list option:

```
keytool –list –storepass zippydoda–keytool -list
```

To remove an entity, use the -delete option:

```
keytool -delete -alias usingjava -storepass abcdefgh
```

Digitally Signing a JAR File:

When you store your applet or application in a JAR file, you can digitally sign the JAR file. If anapplet is loaded from a JAR file signed by a trusted entity, the applet is not subject to the usualsecurity restrictions. In future releases of Java, you will be able to configure what a signedapplet can do, even assigning permissions based on who signed the applet.To sign a JAR file, you first need to add an entry in the keystore database for an alias. After youhave defined the signer, you can use the jarsigner tool to sign the JAR file. To sign thetest.jar file with the alias usingjava that was created in the preceding "Key Management"section, type this:

```
jarsigner -storepass abcdefgh test1.jar usingjava
```

Defining a Policy:

The last piece of the puzzle for signing a JAR file is creating a policy file to define the permissionsto assign to a file. There are two ways to create a permissions file. The first method is tocreate it manually. Listing 8.14 shows how to create a policy file that allows a JAR file signed by"usingjava" to write to a file called newfile

.

Listing 8.14 write.jp—Allow usingjava to Write to the New File

```
grant SignedBy “usingjava” {
permission java.util.PropertyPermission “user.home”, “read”;
permission java.io.FilePermission “${user.home}/newfile”,”write”;
};
```

The Java Security API: Expanding the Boundaries for AppletsThe second way to write a policy is by using the policytool utility bundled with JDK 1.2.Policytool is designed to ease the efforts of defining the policy file.

Running the Applet:

Now that you have signed a JAR file and defined a policy file, you can run the program with the new capabilities. To run the Test program with appletviewer, you can type this:

```
appletviewer -J-Djava.policy=Write.jp file:///test.html
```

The Security API:

The Security API is focused on providing support for digitally signed applets. There is somelevel of support for generating and checking digital signatures from a program, and futureversions will provide classes for encrypting and decrypting information.The Security API exists for two reasons: to allow your programs to perform security functionsand to allowmanufacturers of security software to create their own security provider services.Java 1.1 ships with a single security provider, which is simply called “Sun.” Other vendorsmight provide their own security services. These providers might provide additional servicesbeyond those defined in the Java 1.2 Security API.

Public and Private Key Classes:

The Security API revolves around the manipulation of keys. As you might guess, interfaces are defined for both public and private keys. Because these keys share many common features,they both derive from a common super interface called Key. The three important features of akey are its algorithm, its format, and the encoded key value. You can retrieve these values fromany key by using the following methods:

```
public String getAlgorithm()
public String getFormat()
public byte[] getEncoded()
```

The Signature Class:

The Signature class performs two different roles—it can digitally sign a sequence of bytes, orit can verify the signature of a sequence of bytes. Before you can perform either of these functions,you must create an instance of a Signature class. The constructor for the Signatureclass is protected. The public method for creating signatures is called getInstance, and ittakes the name of the security algorithm and the name of the provider as arguments:

```
public static Signature getInstance(String algorithm,String provider)
```

For the default package provided with Java 1.2, the most common call to getInstance is this:

```
Signature sig = Signature.getInstance(“DSA”, “SUN”)
```

If you are creating a digital signature, call the initSign method in Signature with the private

key you are using to create the signature:

```
public final void initSign(PrivateKey key)
```

If you are verifying a signature, call initVerify with the public key you are verifying against:

```
public final void initVerify(PublicKey key)
```

Whether you are creating a signature or verifying one, you must give the Signature class thesequence of bytes you are concerned with. For instance, if you are digitally signing a file, you must read all the bytes from the file and pass them to the Signature class. The update method allows you to pass data bytes to the Signature class:

```
public final void update(byte b)
public final void update(byte[] b)
```

The update methods are additive; that is, each call to update adds to the existing array of bytesthat will be signed or verified. The following code fragment reads bytes from a file and storesthem in a Signature object:

```
Signature sig = new Signature(“DSA”);
sig.initSign(somePrivateKey);
FileInputStream infile = new FileInputStream(“SignMe”);
int i;
while ((i = infile.read()) >= 0) {
sig.update(i);
}
byte signature[] = sig.sign(); // Do the signing
```

After you have stored the bytes in the Signature, use the sign method to digitally sign them, or use verify to verify them:

```
public final byte[] sign()
public final boolean verify(byte[] otherSignature)
```

Identities and Signers:

As you already know, two types of entities are stored in the key database: identities (publickeys only) and signers (public/private key pairs). The Identity class represents an identity,whereas the Signer class represents a signer. These two classes are abstract classes; you cannotcreate your own instances. Instead, you must go through your security provider to createand locate these classes.When you have an instance of an Identity, you can retrieve its public key with getPublicKeyor set its public key with setPublicKey:

```
public PublicKey getPublicKey()
public void setPublicKey(PublicKey newKey)
```

In addition, you can retrieve all the identity’s certificates using the certificates method:

public Certificate[] certificates()

8.27 THE SECURITY API

You can add and remove certificates with addCertificate and removeCertificate:

public void addCertificate(Certificate cert)

public void removeCertificate(Certificate cert)

The Signer class is a subclass of Identity, and it adds methods for retrieving the private key and setting the key pair:

protected PrivateKey getPrivateKey()

protected final void setKeyPair(KeyPair pair)

Certificates:

A certificate is little more than a digitally signed public key. It also contains the owner of thekey, and the signer. The owner and the signer are called principals, and they are generallyentities that are stored in the key database. You can retrieve the public key from a certificatewith getPublicKey:
public abstract PublicKey getPublicKey()

You can also retrieve the principals from a certificate. The Guarantor is the entity who is signing the public key (guaranteeing its authenticity), and the Principal is the owner of the key that is being guaranteed:

public abstract Principal getPrincipal()

public abstract Principal getGuarantor()

The only interesting method in the Principal interface is getName, which returns the name of the principal:

public abstract String getName()

The IdentityScope Class:

The IdentityScope class represents a set of identities. Generally, this class represents theidentities in the key database. When you have an instance of an IdentityScope, you can addentities, remove entities, and find entities. The getSystemScope method returns the defaultidentity scope for the security system:

public static IdentityScope getSystemScope()

You can locate identities by name, by public key, or using a Principal reference:

public Identity getIdentity(String name)

public Identity getIdentity(PublicKey key)

public Identity getIdentity(Principal principal)

The identities method returns an enumeration that allows you to enumerate through all the identities in the scope:

```
public abstract Enumeration identities()
```

The addIdentity and removeIdentity methods allow you to add new identities to the scope, or to remove old ones:

```
public abstract void addIdentity(Identity id)
public abstract void removeIdentity(Identity id)
```

Listing 8.15 shows a sample program that creates a digital signature for a file and writes the signature to a separate file.

Listing 8.15 Source Code for SignFile.java

```
import java.security.*;
import java.io.*;
import java.util.*;
public class SignFile {
public static void main(String[] args) {
try {
// Get the default identity scope
IdentityScope scope = IdentityScope.getSystemScope();
// Locate the entity named trustme
Identity identity = scope.getIdentity(“usingjava”);
// Create a signature and initialize it for creating a signature
Signature sig = Signature.getInstance(“DSA”, “SUN”);
Signer signer = (Signer) identity;
sig.initSign(signer.getPrivateKey());
// Open the file that will be signed
FileInputStream infile = new FileInputStream(“SignFile.java”);
// Read the bytes from the file and add them to the signature
int i;
while ((i = infile.read()) >= 0) {
sig.update((byte)i);
}
infile.close();
// Open the file that will receive the digital signature of the
// input file
FileOutputStream outfile = new FileOutputStream(
“SignFile.sig”);
// Generate and write the signature
outfile.write(sig.sign());
outfile.close();
} catch (Exception e) {
e.printStackTrace();
}
}
}
```

The capability to generate digital signatures and verify them from a program allows you to provide new levels of security in your programs. This is especially useful in the area of electronic commerce because you can now digitally sign orders and receipts.

8.28 SUMMARY

We have covered Overview of TCP/IP, TCP/IP Protocols, Uniform Resource Locator (URL), Java and URLs.

We have also covered TCP Socket Basics, Creating a TCP Client/Server Application Overview of UDP Messaging, Creating a UDP Server, Creating a UDP Client, Using IP Multicasting.

This chapter also covers The *URL* Class, *URLConnection*, *HTTPURLConnection* Class, *URLEncoder, URLDecoder, URLStreamHandler, ContentHandler class.*

This also covers Socket Class,*ServerSocket, InetAddress, DatagramSocket, DatagramPacket* Class and Multicast Sockets.

This chapter also covers What Necessitates Java Security,The Java Security Framework, Java Security Problems, Java Security API: Expanding the Boundaries for Applets.

8.29 QUESTIONS

1. What are the basics of TCP Socket.
2. How will you Creating a UDP Server and UDP Client.
3. Explain Socket and ServerSocket Class.
4. Explain DatagramSocket and DatagramPacket Class.
5. What Necessitates Java Security?
6. Explain the Java Security Framework.

*****

9

OBJECT SERIALIZATION, REMOTE METHOD SERIALIZATION

Unit Structure

9.1 WHAT IS OBJECT SERIALIZATION?

Up to this point you have been working with objects, and you have learned to create classes so you can manipulate the objects using their methods. However, when you have had to write an object to a different source, say out to a network via a socket or to a file, you have only written out native types like int or char. Object serialization is the tool that was added to Java to allowyou to fully utilize the OOP nature of Java and write those objects you've labored to produce toa file or other stream.

To understand object serialization, first look at an example of how you would go about reading in a simple object, such as a string, from another source. Normally when you open a stream to and from a client program, the odds are fairly good that you are sending/receiving a byte.

You're probably then adding that byte to a string. To do this you might have some code similarto that in Listing 9.1.

Listing 9.1 Notice How Much Work the Computer Has to Do to Generate a

String This Way

```
/* GetString*/
import java.net.*;
import java.io.*;
public class GetString
{
//Read in a String from an URL
public String getStringFromUrl (URL inURL){
InputStream in;
try{
in = inURL.openStream();
} catch (IOException ioe){
System.out.println(“Unable to open stream to URL:”+ioe);
return null;
}
return getString(in);
}
public String getStringFromSocket (Socket inSocket){
InputStream in;
try{
in = inSocket.getInputStream();
} catch (IOException ioe){
System.out.println(“Unable to open stream to Socket:”+ioe);
return null;
}
return getString(in);
}
public String getString (InputStream inStream){
String readString = new String();
DataInputStream in = new DataInputStream (inStream);
char inChar;
try{
while (true){
inChar = (char)in.readByte();
readString = readString + inChar;
}
} catch (EOFException eof){
System.out.println(“The String read was:”+readString);
} catch (IOException ioe) {
System.out.println(“Error reading from stream:”+ioe);
}
return readString;
}
}
```

Most important in Listing 9.1, take a look at the getString() method. Inside of this methodyou will see an indefinitely long while loop (which breaks once an exception is thrown). If youlook closely at what is happening here, you will realize you are reading character-by-charactereach letter in the string and appending it until you reach the end of the file (EOF).

Java has noway without object serialization to actually read in a string as an object..

DataInputStream does have a readLine() which returns a String, but this is not reallythe same for two reasons. First, readLine does not read in an entire file; second, thereadLine() method itself is actually very similar to readString() in Listing 9.1.

n

An even more dire situation arises when you want to read a heterogeneous object such as thatshown in Listing 9.2.

Listing 9.2 A Heterogeneous Object

```
class testObject {
int x;
int y;
float angle;
String name;
public testObject (int x, int y, float angle, String name){
this.x = x ;
this.y = y;
this.angle= angle;
this.name = name;
}
```

What Is Object Serialization?

To read and write testObject without object serialization, you would open a stream, read in a bunch of data, and then use it to fill out the contents of a new object (by passing the read-in elements to the constructor). You might even be able to deduce directly how to read in the first three elements of testObject. But how would you read in the name? Well, because you just wrote a readString class in Listing 9.1 you could use that, but how would you know when the string ends and the next object starts? Even more importantly, what if testObject had even more complicated references? For instance, if testObject looked like Listing 9.3, how would you handle the constant recursion from nextObject?

Listing 9.3 testObject Becomes Even More Complicated

```
class testObject {
int x;
int y;
float angle;
String name;
testObject nextNode;
public testObject (int x, int y, float angle, String name, testObject
nextNode){
this.x = x ;
this.y = y;
this.angle= angle;
this.name = name;
this.nextNode = nextNode;
}
}
```

If you really wanted to, you could write a method (or methods) to read and write Listing 9.3,but wouldn’t it be great if, instead, you could grab an object a whole class at a time?

That’s exactly what object serialization is all about. Do you have a class structure that holds allof the information about a house for a real estate program? No problem—simply open the
stream and send or receive the whole house. Do you want to save the state of a game applet?

Again, no problem. Just send the applet object down the stream.

The ability to store and retrieve whole objects is essential to the construction of all but themost ephemeral of programs. While a full-blown database might be what you need if you’restoring large amounts of data, frequently that’s overkill. Even if you want to implement a database, it would be easier to store objects as BLOB types (byte streams of data) than to break out an int here, a char there, and a byte there.

How Object Serialization Works :

The key to object serialization is to store enough data about the object to be able to reconstructit fully. Furthermore, to protect the user (and programmer), the object must have a “fingerprint”that correctly associates it with the legitimate object from which it was made. This is anaspect not even discussed when looking at writing our own objects. But it is critical for a complete system so that when an object is read in from a stream, each of its fields will be placed
back into the correct class in the correct location.

If you are a C or C++ programmer, you’re probably used to accomplishing much of objectserialization by taking the pointer to a class or struct, doing a sizeOf(), and writing outthe entire class. Unfortunately, Java does not support pointers or direct-memory access, so thistechnique will not work in Java, and object serialization is required.

It’s not necessary, however, for a serialization system to store the methods or the transientfields of a class. The class code is assumed to be available any time these elements are required.

In other words, when you restore the class Date, you are not also restoring the methodgetHours(). It’s assumed that you have restored the values of the Date into a Date object and
that object has the code required for the getHours() method.

Dealing with Objects with Object References :

Objects frequently refer to other objects by using them as class variables (fields). In otherwords, in the more complicated testObject class (refer to Listing 9.3), a nextNode field wasadded. This field is an object referenced within the object. In order to save a class, it is alsonecessary to save the contents of these reference objects. Of course, the reference objects mayalso refer to yet even more objects (such as with testObject if the nextNode also had a validnextNode value). So, as a rule, to serialize an object completely, you must store all of the information for that object, as well as every object that is reachable by the object, including all of the recursive objects.

9.2 OBJECT SERIALIZATION EXAMPLE

As a simple example, store and retrieve a Date class to and from a file. To do this without objectserialization, you would probably do something on the order of getTime() and write the resultinglong integer to the file. However, with object serialization the process is much, much easier.

An Application to Write a Date Class

Listing 9.4 shows an example program called DateWrite. DateWrite creates a Date Object andwrites the entire object to a file.

Object Serialization Example

Listing 9.4 DateWrite.java—An Application that Writes a Date Object to a File

```
import java.io.FileOutputStream;
import java.io.ObjectOutputStream;
import java.util.Date;
public class DateWrite {
public static void main (String args[]){
try{
// Serialize today’s date to a file.
FileOutputStream outputFile = new
FileOutputStream(“dateFile”);
ObjectOutputStream serializeStream = new
åObjectOutputStream(outputFile);
serializeStream.writeObject(“Hi!”);
serializeStream.writeObject(new Date());
serializeStream.flush();
} catch (Exception e) {
System.out.println(“Error during serialization”);
}
}
}//end class DateWrite
```

Take a look at the code in Listing 9.4. First, notice that the program creates aFileOutputStream. In order to do any serialization it is first necessary to declare anoutputStream of some sort to which you will attach the ObjectOutputStream. (As you see inListing 9.5, you can also use the OutputStream generated from any other object, including aURL.)

Once you have established a stream, it is necessary to create an ObjectOutputStream with it.

The ObjectOutputStream contains all of the necessary information to serialize any object andto write it to the stream.In the example of the previous short code fragment, you see two objects being written to thestream. The first object that is written is a String object; the second is the Date object.

To compile Listing 9.4 using the JDK 1.02, you need to add some extra commands thatyou’re probably not used to. Before you do this, though, first verify that you have downloadedthe RMI/object serialization classes and unzipped the file into your Java directory.

Now, type thefollowing command:

```
javac -classpathc:\java\lib\classes.zip;c:\java\lib\objio.zip;. DateWrite.java
```

The previous compiler command assumes you are using a Windows machine and that the directory in which your Java files exist is C:\JAVA. If you have placed it in a different location or are using a system other than Windows, you need to substitute C:\JAVA\LIB with the path that is appropriate for your Java installation. As always, it’s a good idea to take a look at the README file included with your installation, and to read the release notes to learn about any known bugs or problems. This should compile DateWrite cleanly. If you receive an error, though, make sure that you have a OBJIO.ZIP file in your JAVA\LIB directory. Also, make sure that you have included both the CLASSES.ZIP and the OBJIO.ZIP files in your class path. n

Running DateWrite Under JDK 1.02

Once you have compiled the DateWrite program you can run it. However, just as you had to include the OBJIO.ZIP file in the classpath when you compiled the DateWrite class, you must also include it in order to run the class.

```
java -classpath c:\java\lib\classes.zip;c:\java\lib\objio.zip;. DateWrite
```

If you fail to include the OBJIO.ZIP file in your class path, you will likely get an error

such as:

```
java.lang.NoClassDefFoundError: java/io/ObjectOutputStream
at DateWrite.main (DateWrite.java: 9)
```

This is the result of the virtual machine being unable to locate the class files that are required for object serialization.

Compiling and Running DateWrite

To compile and run DateWrite using the JDK 1.1, simply copy the contents of Listing 9.4 to a file called DateWrite and compile it with javac as you would any other file:

```
javac DateWrite.java
```

You can run it just as you would any other Java application as well:

```
java DateWrite
```

No real output is generated by the DateWrite class, so when you run this program you shouldbe returned to the command prompt fairly quickly. However, if you now look in your directorystructure, you should see a file called dateFile. This is the file you just created. If you attemptto type out the file, you will see something that looks mostly like gobbledygook.However, a closer inspection reveals that this file contains several things. The stuff that lookslike gobbledygook is actually what the serialization uses to store information about the class,such as the value of the fields and the class signature that was discussed earlier.

A Simple Application to Read in the Date

The next step, of course, is to read the Date and String back in from the file. See how complicated this could be. Listing 9.5 shows an example program that reads in the String and Date.

Listing 9.5 DateRead.java—An Application that Reads the String and Date Back In

```
import java.io.FileInputStream;
import java.io.ObjectInputStream;
import java.util.Date;
public class DateRead {
public static void main (String args[]){
Date wasThen;
String theString;
try{
// Serialize date from a file.
FileInputStream inputFile = new FileInputStream(“dateFile”);
ObjectInputStream serializeStream = new
ObjectInputStream(inputFile);
theString = (String) serializeStream.readObject();
} catch (Exception e) {
System.out.println(“Error during serialization”);
return;
}
System.out.println(“The string is:”+theString);
System.out.println(“The old date was:”+wasThen);
}
}
```

Listings 9.4 and 9.5 differ primarily in the ways that you would expect. Listing 9.4 is writing,and Listing 9.5 is reading. In DateRead, you first declare two variables to store the objects in.You need to remember to do this because if you were to create the variables inside the trycatchblock, they would go out of scope before reaching the System.out line. Next, aFileInputStream and ObjectInputStream are created, just as the FileOutputStream and ObjectOutputStreams were created for DateWrite.

The next two lines of the code are also probably fairly obvious, but pay special attention to thecasting operator. readObject() returns an Object class. By default, Java does not polymorphcastany object, so you must implicitly direct it to do so. The rest of the code should be fairlyobvious to you by now.

You can compile and run DateRead, so simply follow the same directions for DateWrite.

To compile the code using JDK 1.02, this time set a classpath variable so that you don’talways have to use the -classpath option with javac. You can use the –classpathoption as done in the previous example, but this solution is a bit more efficient. In either case these solutions are interchangeable. To set the classpath this way do the following:

On a Windows machine, type:

```
set classpath=c:\java\lib\classes.zip;c:\java\lib\objio.zip;.
```

On other platforms, the syntax is slightly different. For instance, under UNIX you might type:

```
classpath=/usr/java/lib/classes.zip:/usr/java/lib/objio.zip:.
export classpath
```

In either case, don’t forget to add the current directory (.) to the end of the classpath statement.

Javac will run without the current directory being listed, but the java command won’t work. n

Compiling and Running DateRead

You can compile and run DateRead, simply by following the same

directions for DateWrite.

```
javac DateRead.java
```

You can also run it by using the familiar java command:

```
java DateRead
```

Here’s an example of the resulting output from this code:

```
The String is:Hi!
The old date was:Wed Dec 1 23:36:26 edt 1996
```

Notice that the String and Date are read in just as they were when you wrote them out. Now you can write out and read entire objects from the stream without needing to push each element into the stream.

Listing 9.6 shows DateRead changed so that it can also be run as an applet.

Listing 9.6 DataReadApp.java—An Applet That Reads a Date Object to a File

```
import java.io.FileInputStream;
import java.io.ObjectInputStream;
import java.util.Date;
import java.awt.Graphics;
public class DateReadApp extends java.applet.Applet {
public void paint (Graphics g){
Date wasThen;
String theString;
try{
// Serialize date from a file.
FileInputStream inputFile = new FileInputStream(“dateFile”);
ObjectInputStream serializeStream = new
ObjectInputStream(inputFile);
theString = (String) serializeStream.readObject();
wasThen = (Date)serializeStream.readObject();
} catch (Exception e) {
System.out.println(“Error during serialization”);
return;
}
g.drawString((“The string is:”+theString),5,100);
g.drawString((“The old date was:”+wasThen),5,150);
}
}
```

Object Serialization Example

After you have compiled Listing 9.6, the resulting output should look like Figure 9.1. Remember

that you will have to use Applet Viewer to run this applet, because other browsers don’t yet

support object serialization.

9.3 WRITING AND READING YOUR OWN OBJECTS

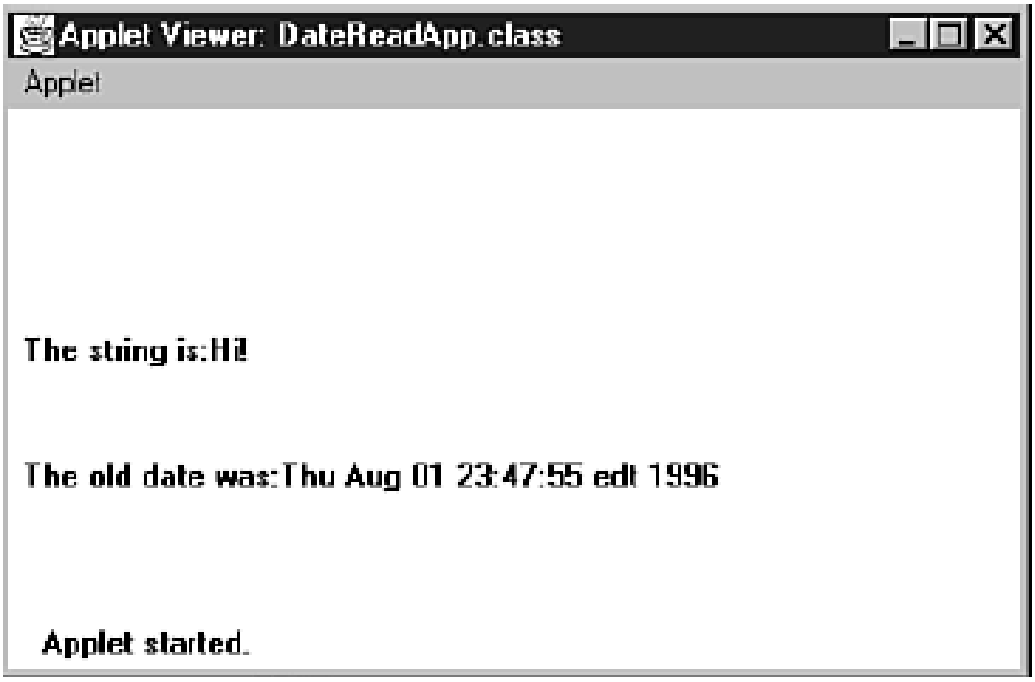
By default, you have the ability to write and read most of your own objects, just as you did withthe Date class. There are certain restrictions right now (such as if the object refers to a nativepeer), but for the most part, any class that you create can be serialized.

Listings 9.7 through 9.9 show the source code for serializing an example class calledSerializeObject.

Listing 9.7 SerializeObject—A Simple Class with a Couple of Fields

```
public class SerializeObject implements java.io.Serializable{
public int first;
public char second;
public String third;
public SerializeObject (int first, char second, String third){
this.first= first;
this.second = second;
this.third = third;
}
}
```

FIG. 9.1 The Date and String have been read in using serialization.



```
The string is:Hi!

The old date was:Thu Aug 01 23:47:55 edt 1996

Applet started.
```

Listing 9.8 ObjWrite—Write Out a Sample SerializeObject to a File

```
import java.io.FileOutputStream;
import java.io.ObjectOutputStream;
import SerializeObject;
public class ObjWrite {
public static void main (String args[]){
try{
FileOutputStream outputFile = new FileOutputStream(“objFile”);
ObjectOutputStream serializeStream = new
ObjectOutputStream(outputFile);
SerializeObject obj = new SerializeObject (1,’c’,new String
(“Hi!”));
serializeStream.writeObject(obj);
serializeStream.flush();
} catch (Exception e) {
System.out.println(“Error during serialization”);
}
}
}
```

Listing 9.9 ObjRead—Read in the Same Object from the File

```
import java.io.FileInputStream;
import java.io.ObjectInputStream;
import SerializeObject;
public class ObjRead extends java.applet.Applet {
```

```
public void init(){
main(null);
}
public static void main (String args[]){
SerializeObject obj;
try{
FileInputStream inputFile = new FileInputStream(“objFile”);
ObjectInputStream serializeStream = new
ObjectInputStream(inputFile);
obj = (SerializeObject)serializeStream.readObject();
} catch (Exception e) {
System.out.println(“Error during serialization”);
return;
}
System.out.println(“first is:”+obj.first);
System.out.println(“second is:”+obj.second);
System.out.println(“third is:”+obj.third);
}
}
```

Writing and Reading Your Own Objects

NOTE

In the previous example classes, notice that the SerializeObject class refers to a number of things, including another class—String. As you might already suspect, once you have compiled and run each of these classes, the resulting output is

```
First is:1
Second is:c
Third is:Hi!
```

What’s most amazing about all this code is how easy it is to transfer the object.

9.4 CUSTOMIZING OBJECT SERIALIZATION

Sometimes it is useful, or even necessary, to control how an individual object is serialized. If forinstance you want to encrypt the data held by this object in a proprietary form to control accessto the objects data, you would not want to use the default serialization mechanisms.

Special Serialization changed in JDK 1.2. If you are using a previous version of the JDK, themethods shown here will not work for you. Instead, you should refer to the API to learn how to change the serialization for your objects. Look for the interfaces java.io.Replaceable andjava.io.Resolvable.

To override how an object is serialized, you must define two methods in your class with thesignatures:

```
private void writeObject(java.io.ObjectOutputStream out)
throws IOException
private void readObject(java.io.ObjectInputStream in)
throws IOException, ClassNotFoundException;
```

The first question you’re probably asking yourself at this point is, if writeObject() and readObject() are not in the Serializable interface, how does the serialization system manage to call these methods? The answer is that it uses what is know as Reflection. Reflection is covered in Chapter , “Reflection,” but essentially it allows programs to access methods and constructors of components based on knowing their signature. Reflection is generally a complicated API, and for most of your programs you will not need to be concerned with actually getting Reflection to work. However, you do need to know that Reflection requires the signaturesof the methods to be exact. Therefore, it is critical that you use exactly these signatures. Failureto make the methods private will cause the serialization mechanism to use it’s default algorithms.

Your classes do not need to be concerned with calling super.writeObject() orsuper.readObject(), nor do you need to be concerned about how subclasses will serialize the class as each portion of the object will be handled separately by the serialization mechanism.

On the other hand, if you want to use the default mechanism within the writeObject()method, you can do so by calling out.defaultWriteObject(). Or from the readObject() method you can call in.defaultReadObject().

Listing 9.10 contains a class called DateTest that writes out the value of a date as three separateintegers—the year, the month, and the day of the month—instead of using the default serialization. Listings 9.11 and 9.12 contain sample classes for testing the DateTest class.

Listing 9.10 DateTest—A Class with Special Serialization

```
import java.io.*;
import java.util.*;
import java.text.*;
public class DateTest implements Serializable{
transient GregorianCalendar myDate;
public void newDate(){
myDate = new GregorianCalendar();
}
private void writeObject(ObjectOutputStream out) throws
IOException{
int year = myDate.get(Calendar.YEAR);
int month = myDate.get(Calendar.MONTH);
int day = myDate.get(Calendar.DAY_OF_MONTH);
out.writeInt(year);
out.writeInt(month);
out.writeInt(day);
}
private void readObject(ObjectInputStream in) throws
IOException,
åClassNotFoundException{
int year = in.readInt();
int month = in.readInt();
int day = in.readInt();
myDate = new GregorianCalendar(year,month,day);
}
public String toString(){
DateFormat df = DateFormat.getDateInstance();
return “DateTest:”+df.format(myDate.getTime());
}
}
```

Listing 9.11 DateWriter—A Class That Writes Out a DateTest

```
import java.io.*;
public class DateWriter{
public static void main(String args[]){
try{
DateTest test = new DateTest();
test.newDate();
System.out.println(“Writting test:”+test);
```

continued

Customizing Object Serialization

```
FileOutputStream fout = new FileOutputStream(“test.out”);
ObjectOutputStream oout = new ObjectOutputStream (fout);
oout.writeObject(test);
}catch (Exception ioe){
ioe.printStackTrace(System.err);
}
}
}
```

Listing 9.12 DateReader—A Class That Reads in a Datetest

```
import java.io.*;
public class DateReader{
public static void main(String args[]){
```

```
try{
FileInputStream fin = new FileInputStream(“test.out”);
ObjectInputStream oin = new ObjectInputStream (fin);
DateTest test = (DateTest)oin.readObject();
System.out.println(“Read dateTest as “+test);
}catch (Exception e){
e.printStackTrace(System.err);
}
}
}
```

9.5 WHAT IS REMOTE METHOD INVOCATION (RMI) ?

Remote Method Invocation (RMI) allows a Java object that executes on one machineto invoke a method of a Java object that executes on another machine. This is animportant feature, because it allows you to build distributed applications. While acomplete discussion of RMI is outside the scope of this book, the following exampledescribes the basic principles involved.

9.6A SIMPLE CLIENT/SERVER APPLICATION USING RMI

This section provides step-by-step directions for building a simple client/serverapplication by using RMI. The server receives a request from a client, processes it, andreturns a result. In this example, the request specifies two numbers. The server addsthese together and returns the sum.

Step One: Enter and Compile the Source Code

This application uses four source files. The first file, **AddServerIntf.java**, defines theremote interface that is provided by the server. It contains one method that acceptstwo **double** arguments and returns their sum. All remote interfaces must extend the**Remote** interface, which is part of **java.rmi**. **Remote** defines no members. Its purposeis simply to indicate that an interface uses remote methods. All remote methods canthrow a **RemoteException.**

```
import java.rmi.*;
public interface AddServerIntf extends Remote {
double add(double d1, double d2) throws RemoteException;
}
```

The second source file, **AddServerImpl.java**, implements the remote interface.

The implementation of the **add( )** method is straightforward. All remote objects must extend **UnicastRemoteObject**, which provides functionality that is needed to make objects available from remote machines.

```
import java.rmi.*;
import java.rmi.server.*;
public class AddServerImpl extends UnicastRemoteObject
implements AddServerIntf {
public AddServerImpl() throws RemoteException {
}
public double add(double d1, double d2) throws RemoteException {
return d1 + d2;
}
}
```

The third source file, **AddServer.java**, contains the main program for the servermachine. Its primary function is to update the RMI registry on that machine. This isdone by using the **rebind ( )** method of the **Naming** class (found in **java.rmi**). Thatmethod associates a name with an object reference. The first argument to the **rebind( )**method is a string that names the server as “AddServer”. Its second argument is areference to an instance of **AddServerImpl**.

```
import java.net.*;
import java.rmi.*;
public class AddServer {
public static void main(String args[]) {
try {
AddServerImpl addServerImpl = new AddServerImpl();
Naming.rebind("AddServer", addServerImpl);
}
catch(Exception e) {
System.out.println("Exception: " + e);
}
}
}
```

The fourth source file, **AddClient.java**, implements the client side of thisdistributed application. **AddClient.java** requires three command line arguments. Thefirst is the IP address or name of the server machine. The second and third argumentsare the two numbers that are to be summed.

The application begins by forming a string that follows the URL syntax. This URLuses the **rmi** protocol. The string includes the IP address or name of the server and thestring “AddServer”. The program then invokes the **lookup( )** method of the **Naming**class. This method accepts one argument, the **rmi** URL, and returns a reference to anobject of type **AddServerIntf**. All remote method invocations can then be directed tothis object.

The program continues by displaying its arguments and then invokes the remote**add( )** method. The sum is returned from this method and is then rinted.

```
import java.rmi.*;
public class AddClient {
public static void main(String args[]) {
try {
String addServerURL = "rmi://" + args[0] + "/AddServer";
AddServerIntf addServerIntf =
(AddServerIntf)Naming.lookup(addServerURL);
System.out.println("The first number is: " + args[1]);

double d1 = Double.valueOf(args[1]).doubleValue();
System.out.println("The second number is: " + args[2]);
double d2 = Double.valueOf(args[2]).doubleValue();
System.out.println("The sum is: " + addServerIntf.add(d1, d2));
}
catch(Exception e) {
System.out.println("Exception: " + e);
}
}
}
```

After you enter all the code, use **javac** to compile the four source files that you created.

Step Two: Generate Stubs and Skeletons:

Before you can use the client and server, you must generate the necessary stub. Youmay also need to generate a skeleton. In the context of RMI, a *stub* is a Java object thatresides on the client machine. Its function is to present the same inerfaces as theremote server. Remote method calls initiated by the client are actually directed to thestub. The stub works with the other parts of the RMI system to formulate a request thatis sent to the remote machine.

A remote method may accept arguments that are simple types or objects. In thelatter case, the object may have references to other objects. All of this information mustbe sent to the remote machine. That is, an object passed as an argument to a remotemethod call must be serialized and sent to the remote machine. Recall from Chapterthat the serialization facilities also recursively process all referenced objects.

Skeletons are not required by Java 2. However, they are required for the Java 1.1RMI model. Because of this, skeletons are still required for compatibility between Java1.1 and Java 2. A *skeleton* is a Java object that resides on the server machine. It workswith the other parts of the 1.1 RMI system to receive requests, perform deserialization,and invoke the appropriate code on the server. Again, the skeleton mechanism is notrequired for Java 2 code that does not require compatibility with 1.1. Because manyreaders will want to generate the skeleton, one is used by this example.

If a response must be returned to the client, the process works in reverse. Notethat the serialization and deserialization facilities are also used if objects are returnedto a client.To generate stubs and skeletons, you use a tool called the *RMI compiler,* which isinvoked from the command line, as shown here:

```
rmic AddServerImpl
```

This command generates two new files: **AddServerImpl_Skel.class** (skeleton) and**AddServerImpl_Stub.class** (stub). When using **rmic**, be sure that **CLASSPATH** is setto include the current directory. As you can see, by default, **rmic** generates both a stuband a skeleton file. If you do not need the skeleton, you have the option to suppress it.Step Three: Install Files on the Client and Server Machines

Copy **AddClient.class**, **AddServerImpl_Stub.class**, and **AddServerIntf.class** to adirectory on the client machine. Copy **AddServerIntf.class**, **AddServerImpl.class**,**AddServerImpl_Skel.class**, **AddServerImpl_Stub.class**, and **AddServer.class** to adirectory on the server machine.

RMI has techniques for dynamic class loading, but they are not used by the example athand. Instead, all of the files that are used by the client and server applications must beinstalled manually on those machines.

Step Four: Start the RMI Registry on the Server Machine:

The Java 2 SDK provides a program called **rmiregistry**, which executes on the servermachine. It maps names to object references. First, check that the **CLASSPATH**environment variable includes the directory in which your files are located.

Then,start the RMI Registry from the command line, as shown here:

```
start rmiregistry
```

When this command returns, you should see that a new window has been created.

You need to leave this window open until you are done experimenting with the
RMI example.

Step Five: Start the Server :

The server code is started from the command line, as shown here:

```
java AddServer
```

Recall that the **AddServer** code instantiates **AddServerImpl** and registers that objectwith the name “AddServer”.

Step Six: Start the Client :

The **AddClient** software requires three arguments: the name or IP address of the servermachine and the two numbers that are to be summed together. You may invoke it fromthe command line by using one of the two formats shown here:

```
java AddClient server1 8 9
java AddClient 11.12.13.14 8 9
```

In the first line, the name of the server is provided. The second line uses its IP address
(11.12.13.14).

You can try this example without actually having a remote server. To do so, simplyinstall all of the programs on the same machine, start **rmiregistry**, start **AddSever**, andthen execute **AddClient** using this command line:

```
java AddClient 127.0.0.1 8 9
```

Here, the address 127.0.0.1 is the “loop back” address for the local machine. Using this
address allows you to exercise the entire RMI mechanism without actually having to
install the server on a remote computer.

In either case, sample output from this program is shown here:

```
The first number is: 8
The second number is: 9
The sum is: 17.0
```

9.7 SUMMARY

This chapter covers what Is Object Serialization with its example, how to write and read our Own Objects, Customizing Object Serialization.

It covers what is Remote Method Invocation and a simple Client/Server Application Using RMI.

9.8 QUESTIONS

1. What Is Object Serialization?
2. Give an examples Object Serialization
3. How will you write and read your Own Objects .
4. How will you Customize Object Serialization.
5. What Is Remote Method Invocation?
6. Writea source code demonstrate a Simple Client/Server Application Using RMI.

*****

10

JDBC: THE JAVA DATABASE CONNECTIVITY

Unit Structure

10.1 JDBC OVERVIEW

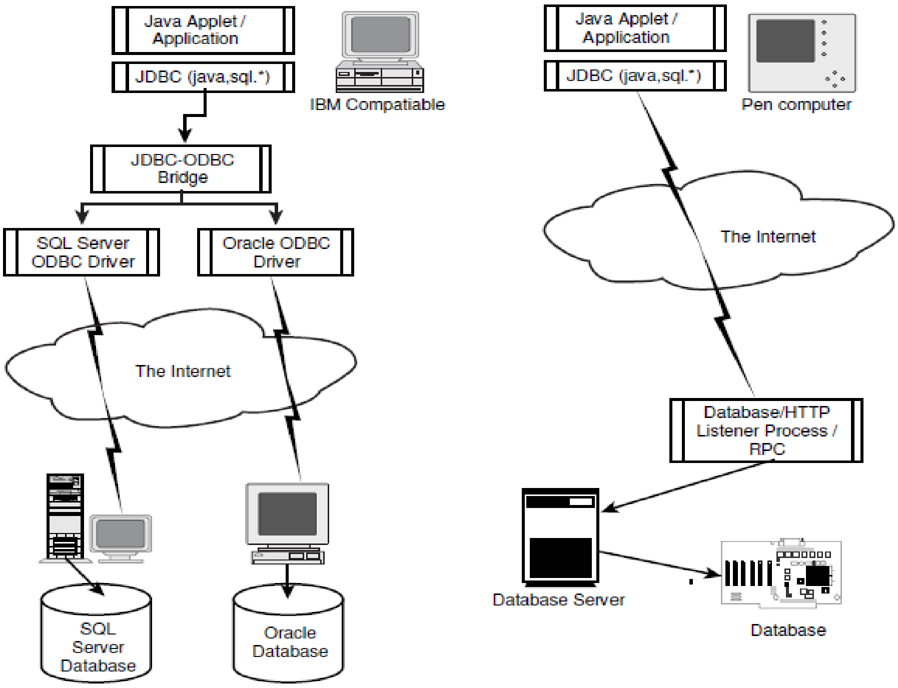
JDBC is a Java Database Connectivity API that is a part of the Java Enterprise APIs from Sun Microsystems, Inc. From a developer's point of view, JDBC is the first standardized effort to integrate relational databases with Java programs. JDBC has opened all the relational power that can be mustered to Java applets and applications. In this chapter and the next, you take an in-depth look at the JDBC classes and methods. Java Database Connectivity is a set of relational database objects and methods for interacting with SQL data sources. The JDBC APIs are part of the Enterprise APIs of Java 1.1 and, thus, are a part of all Java Virtual Machine (JVM) implementations.

How Does JDBC Work?

As previously discussed, JDBC is designed on the CLI model. JDBC defines a set of API objectsand methods to interact with the underlying database. A Java program first opens a connection to a database, makes a Statement object, passes SQL statements to the underlying DBMS through the Statement object, and retrieves the results as well as information about the result sets. Typically, the JDBC class files and the Java applet reside in the client. They can be downloaded from the network also. To minimize the latency during execution, it is better to have the JDBC classes in the client. The DBMS and the data source are typically located in a remote server.

Figure 10.1 shows the JDBC communication layer alternatives. The applet and the JDBC layers communicate in the client system, and the driver takes care of interacting with the database over the network. FIG. 10.1 JDBC communication layer alternatives: The JDBC driver can be a native library, like the JDBC-ODBC bridge, or a Java class talking across the network to an RPC or HTTP listener process in the database server.

Figure 10.1



The JDBC classes are in the java.sql package, and all Java programs use the objects and methods in the java.sql package to read from and write to data sources. A program using the JDBC will need a driver for the data source with which it wants to interface. This driver can be a native module (like the JDBCODBC.DLL for the Windows JDBC-ODBC bridge developed by Sun/Intersolv), or it can be a Java program that talks to a server in the network by using some RPC or an HTTP talker-listener protocol. Both schemes are shown in Figure 10.1. It is conceivable that an application will deal with more than one data source—possibly heterogeneous data sources. (A database gateway program is a good example of an application that accesses multiple heterogeneous data sources.) For this reason, JDBC has a DriverManager

FIG. 10.1 JDBC communication layer alternatives:

The JDBC driver can be a native library, like the JDBC-ODBC bridge, or a Java class talking across the network to an RPC or HTTP listener process in the database server.

JDBC Overview :

whose function is to manage the drivers and provide a list of currently loaded drivers to the application programs.

Security Model :

Security is always an important issue, especially when databases are involved. As of the writing of this book, JDBC follows the standard security model in which applets can connect only to the server from where they are loaded; remote applets cannot connect to local databases. Applications have no connection restrictions. For pure Java drivers, the security check is automatic. For drivers developed in native methods, however, the drivers must have some security checks.

JDBC-ODBC Bridge :

As a part of JDBC, Sun also delivers a driver to access ODBC data sources from JDBC. This driver is jointly developed with Intersolv and is called the JDBC-ODBC bridge. The JDBCODBC bridge is implemented as the JdbcOdbc .class and a native library to access the ODBC driver. For the Windows platform, the native library is a DLL (JDBCODBC.DLL).

Because JDBC is close to ODBC in design, the ODBC bridge is a thin layer over JDBC. Internally, this driver maps JDBC methods to ODBC calls and, thus, interacts with any available ODBC driver. The advantage of this bridge is that now JDBC has the capability to access almost all databases, as ODBC drivers are widely available. You can use this bridge (Version1.2001) to run the sample programs in this and the next chapter

10.2 JDBC IMPLEMENTATION

JDBC is implemented as the java.sql package. This package contains all the JDBC classes and methods, as shown in Table 10.1.

Table 10.1 J DBC Classes

Type	Class
Driver	Java.sql.Driver Java.sql.Driver.Manager Java.sql .Driver Property Info
Connection	Java.sql.Connection
Statements	Java.sql.Statement Java.sql.PreparedStatement Java.sql.CallableStatement

Resultset	Java.sql.ResultSet
Errors/Warning	Java.sql.Sql.SQLException Java.sql. SQLWarning
Metadata	Java.sql.DatabaseMetadata Java.sql.ResultSetMetadata
Date/Time	Java.sql.Date Java.sql.Time Java.sql.Time stamp
Miscellaneous	Java.sql.Types Java.sql.DataTruncation

Now look at these classes and see how you can develop a simple JDBC application.

JDBC Classes—Overview:

When you look at the class hierarchy and methods associated with it, the topmost class in the hierarchy is the DriverManager.The DriverManager keeps the driver information, state information, and more. When each driver is loaded, it registers with the DriverManager. The DriverManager, when required to open a connection, selects the driver depending on the JDBC URL.JDBC URL true to the nature of the Internet, JDBC identifies a database with an URL. The URL's form is asfollows:

jdbc:<subprotocol>:<subname related to the DBMS/Protocol>
For databases on the Internet or intranet, the subname can contain the Net URL //hostname:port/.

The <subprotocol> can be any name that a database understands. The odbc subprotocol name is
reserved for ODBC-style data sources. A normal ODBC database JDBC URL looks like the following:

jdbc:odbc:<ODBC DSN>;User=<username>;PW=<password>

If you are developing a JDBC driver with a new subprotocol, it is better to reserve the subprotocolname with Sun, which maintains an informal subprotocol registry.The java.sql.Driver class is usually referred to for information such as PropertyInfo, version number, and so on. This class could be loaded many times during the execution of a Java program using the JDBC API.

Looking at the java.sql.Driver and java.sql.DriverManager classes and methods as listed in Table 10.2, you see that the DriverManager returns a Connection object when you use the getConnection() method.

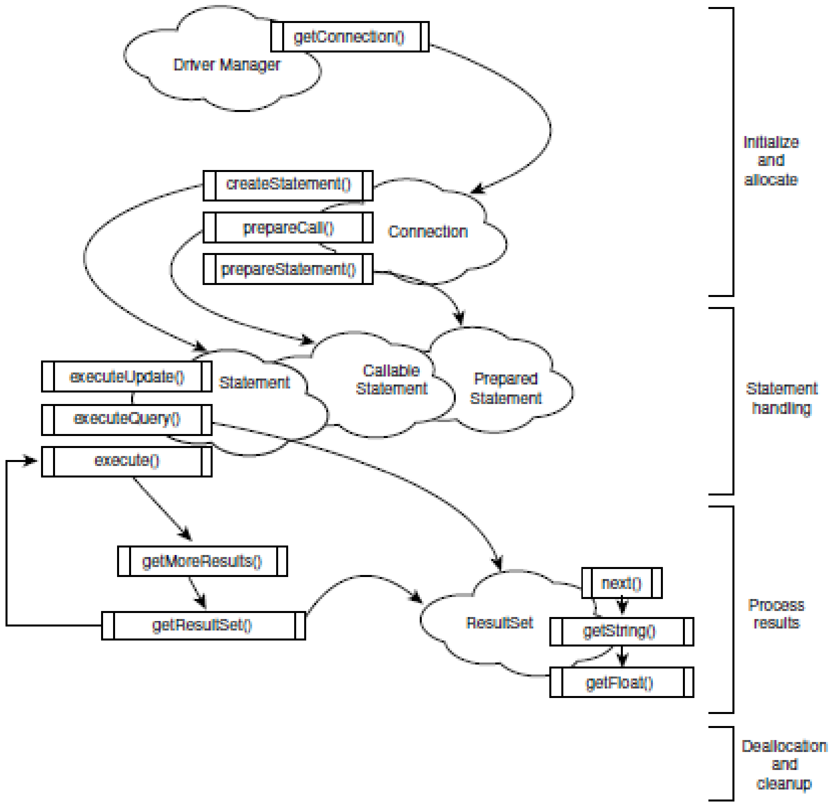
Table 10.2 Driver, DriverManager, and Related Methods

Return Type	Method Name	Parameter
java.sql.Driver		
Connection	connect	(String url, java.util.Propertiesinfo)
Boolean	acceptsURL	(String url)
DriverPropertyInfo[]	getPropertyInfo	(String url, java.util.Propertiesinfo)
int	getMajorVersion	()
Int	getMinorVersion	()
Boolean	jdbcCompliant	()
java.sql.DriverManager		
Connection	getConnection	(String url, java.util.Propertiesinfo)
Connection	getConnection	(String url, String user, Stringpassword)
Connection	getConnection	(String url)
Driver	getDriver	(String url)
Void	registerDriver	(java.sql.Driver driver)
Void	deregisterDriver	(Driver driver)
java.util.Enumeration	getDrivers ()	
void	setLoginTimeout	(int seconds)
int	getLoginTimeout	()
void	setLogStream	(java.io.PrintStream out)
java.io.PrintStream	getLogStream	()
void	println	(String message)
Class Initialization Routine		
Void	initialize	()

Other useful methods include the registerDriver(), deRegister(), and getDrivers() methods. By using the getDrivers() method, you can get a list of registered drivers. Figure 10.2shows the JDBC class hierarchy, as well as the flow of a typical Java program using the JDBCAPIs. In the following section, you will follow the steps required to access a simple database by using JDBC and the JDBC-ODBC driver. Anatomy of a JDBC Application .To handle data from a database, a Java program follows these general steps. (Figure 10.2shows the general JDBC objects, the methods, and the sequence.) First, the program calls the getConnection() method to get the Connection object. Then it creates the Statement object and prepares a SQL statement. A SQL statement can be executed immediately (Statement object), can be a compiled statement (PreparedStatement object), or can be a call to a stored procedure (CallableStatement object).When the method executeQuery() is executed, a ResultSet object is returned. SQL statements such as update or delete will not return a ResultSet. For such statements, the executeUpdate() method is used. The executeUpdate() method returns an integer that denotes the number of rows affected by the SQL statement. The ResultSet contains rows of data that are parsed using the next() method. In case of a transaction processing application, methods such as rollback() and commit() can be used either to undo the changes made by the SQL statements or permanently affect the changes made by the SQL statements.

Figure 10.2

JDBC class hierarchy and a JDBC API flow.



JDBC Examples:

These examples access the Student database, the schema of which is shown in Figure 10.3.The tables in the examples that you are interested in are the Students table, Classes table, Instructors table, and Students_Classes table. This database is a Microsoft Access database. The full database and sample data are generated by the Access Database Wizard. You access the database by using JDBC and the JDBC-ODBC bridge. Before you jump into writing a Java JDBC program, you need to configure an ODBC datasource. As you saw earlier, the getConnection() method requires a data source name (DSN),user ID, and password for the ODBC data source. The database driver type or subprotocol name is odbc. So the driver manager finds out from the ODBC driver the rest of the details.But wait, where do you put the rest of the details? This is where the ODBC setup comes into the picture. The ODBC Setup program runs outside the Java application from the

Microsoft ODBC program group. The ODBC Setup program enables you to set up the data source so that this information is available to the ODBC Driver Manager, which in turn loads the Microsoft Access ODBC driver. If the database is in another DBMS form—say, Oracle—you configure this source as Oracle ODBC driver. In Windows 3.x, the Setup program puts this information in the ODBC.INI file. With Windows 95 and Windows NT 4.0, this information is in the Registry. Figure 10.3 shows the ODBC Setup screen.

Figure 10.3 JDBC example database schema.

.

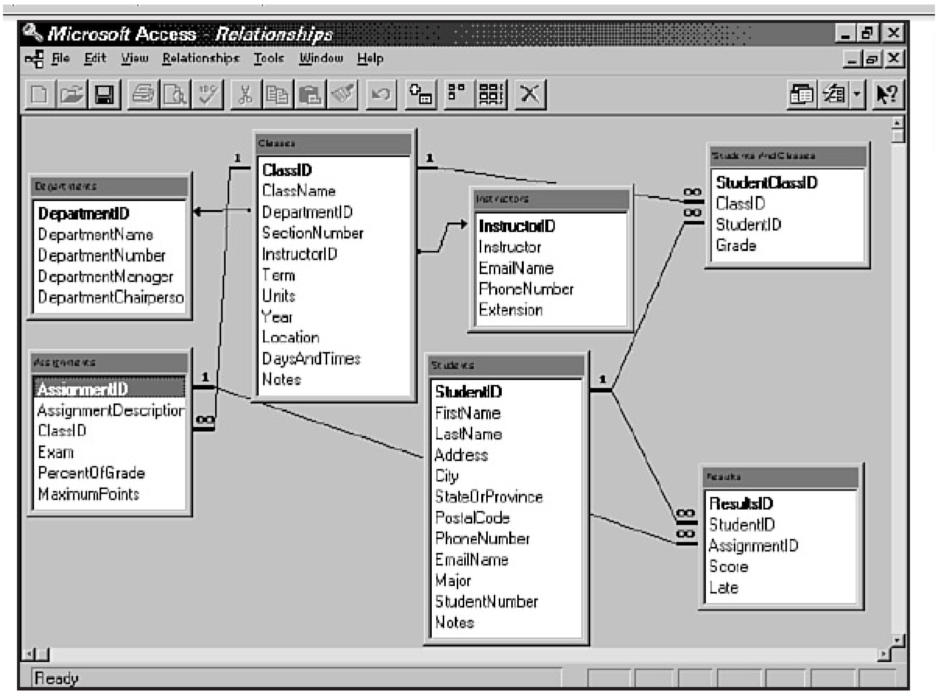
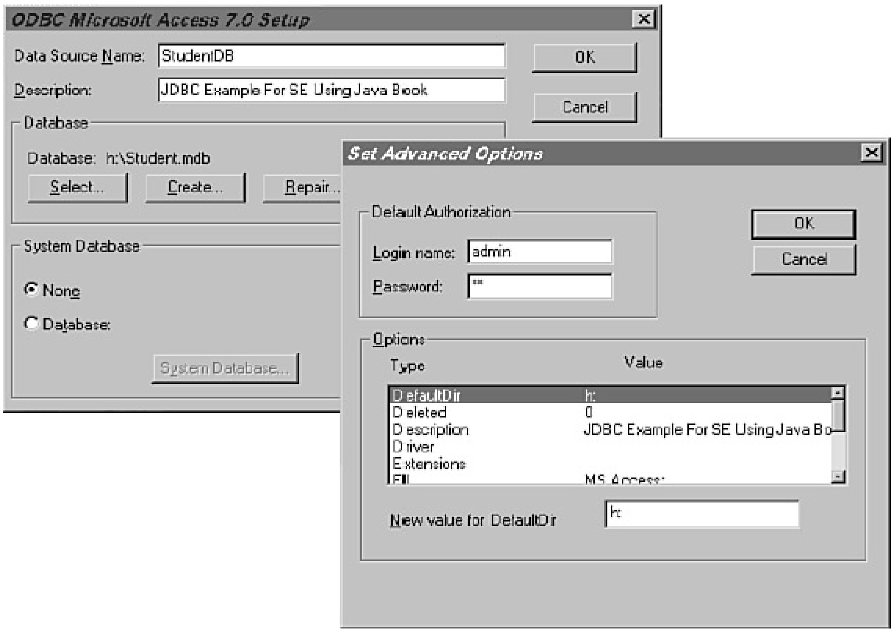


Figure 10.4 ODBC Setup for the example database. After this setup, the example database URL is jdbc:odbc: Student DB;uid= ”admin”;pw=”sa”.



Data Source Name: StudentDB

Description: JDBC Example For SE Using Java Book

Database

Database: h:\Student.mdb

System Database

● None

○ Database:

Default Authorization

Login name: admin

Password: **

Options

Type	Value
DefaultDir	h:
Deleted	0
Description	JDBC Example For SE Using Java Bo
Driver	
Extensions	
Fil	MS Access;

New value for DefaultDir: h:

JDBC Query Example In this example, you list all the students in the database with a SQL SELECT statement. The steps required to accomplish this task using the JDBC API are listed here. For each step, the Java program code with the JDBC API calls follows the description of the steps.

```
//Declare a method and some variables.
public void ListStudents() throws SQLException {
int i, NoOfColumns;
String StNo,StFName,StLName;
//Initialize and load the JDBC-ODBC driver.
Class.forName (“jdbc.odbc.JdbcOdbcDriver”);
//Make the connection object.
Connection Ex1Con =
DriverManager.getConnection(“jdbc:odbc:StudentDB;uid=”admi
n”;pw=”sa”);
//Create a simple Statement object.
Statement Ex1Stmt = Ex1Con.createStatement();
//Make a SQL string, pass it to the DBMS, and execute the SQL
statement.
ResultSet Ex1rs = Ex1Stmt.executeQuery(
“SELECT StudentNumber, FirstName, LastName FROM
Students”);
//Process each row until there are no more rows.
// Displays the results on the console.
System.out.println(“Student Number First Name Last Name”);
while (Ex1rs.next()) {
// Get the column values into Java variables
StNo = Ex1rs.getString(1);
StFName = Ex1rs.getString(2);
StLName = Ex1rs.getString(3);
System.out.println(StNo,StFName,StLName);
}
}
```

As you can see, it is a simple Java program using the JDBC API. The program illustrates the basic steps needed to access a table and lists some of the fields in the records. JDBC Update Example In this example, you update the FirstName field in the Students tableby knowing the student’s StudentNumber. As in the preceding example, the code follows the description of the step.

```
//Declare a method and some variables and parameters.
public void UpdateStudentName(String StFName, String
StLName,
String StNo) throws SQLException {
int RetValue;
// Initialize and load the JDBC-ODBC driver.
Class.forName (“jdbc.odbc.JdbcOdbcDriver”);
// Make the connection object.
Connection Ex1Con = DriverManager.getConnection
 ( “jdbc:odbc:StudentDB;uid=”admin”;pw=”sa”);
// Create a simple Statement object.
Statement Ex1Stmt = Ex1Con.createStatement();
//Make a SQL string, pass it to the DBMS, and execute the SQL statement
String SQLBuffer = “UPDATE Students SET FirstName = “+
StFName+”, LastName = “+StLName+
“ WHERE StudentNumber = “+StNo
RetValue = Ex1Stmt.executeUpdate( SQLBuffer);
System.out.println(“Updated “ + RetValue + “ rows in the Database.”);
}
```

In this example, you execute the SQL statement and get the number of rows affected by the SQL statement back from the DBMS.

The previous two examples show how you can do simple yet powerful SQL manipulation of the underlying data by using the JDBC API in a Java program. In the following sections, you examine each JDBC class in detail.

10.3 THE CONNECTION CLASS

The Connection class is one of the major classes in JDBC. It packs a lot of functionality, ranging from transaction processing to creating statements, in one class as seen in Table 10.3.

Table 10.3 java.sql. Connection Methods and Constants Return Type Method Name Parameter

Return Type	**Method Name**	**Parameter**
Statement-Related Methods		
Statement	createStatement	()
PreparedStatement	prepareStatement	(String sql)
CallableStatement	prepareCall	(String sql)
String	nativeSQL	(String sql)
void	close	()
boolean	isClosed	()
Metadata-Related Methods		
DatabaseMetaData	getMetaData	()
void	setReadOnly	(boolean readOnly)
boolean	isReadOnly	()
void	setCatalog	(String catalog)

String	getCatalog	()
SQLWarning	getWarnings	()
void	clearWarnings	()
Transaction-Related Methods		
void	setAutoCommit	(boolean autoCommit)
boolean	getAutoCommit	()
void	commit	()
void	rollback	()
void	setTransaction Isolation	(int level)
int	getTransaction Isolation	()

The Transaction Isolation constants are defined in the java.sql.Connection as integers with the following values:
Transaction Isolation Constant Name Value

TRANSACTION_NONE 0
TRANSACTION_READ_UNCOMMITTED 1
TRANSACTION_READ_COMMITTED 2
TRANSACTION_REPEATABLE_READ 4
TRANSACTION_SERIALIZABLE 8

As you saw earlier, the connection is for a specific database that can be interacted with in a specific subprotocol. The Connection object internally manages all aspects about a connection, and the details are transparent to the program. Actually, the Connection object is a pipeline into the underlying DBMS driver. The information to be managed includes the data source identifier, the subprotocol, the state information, the DBMS SQL execution plan ID or handle, and any other contextual information needed to interact successfully with the underlying DBMS. The data source identifier could be a port in the Internet database server that is identified by the //<server name>:port/ URL or just a data source name used by the ODBCdriver or a full pathname to a database file in the local computer. For all you know, it could be a pointer to data feed of the stock market prices from Wall Street. Another important function performed by the Connection object is the transaction management. The handling of the transactions depends on the state of an internal autocommit flag that is set using the setAutoCommit() method, and the state of this flag can be read using the getAutoCommit() method. When the flag is true, the transactions are automatically committed as soon as they are completed. There is no need for any intervention or commands from the Java application program. When the flag is false, the system is in the Manual mode. The Java program has the option to commit the set of transactions that happened after the last commit or roll back the transactions using the commit() and rollback() methods. JDBC also provides methods for setting the transaction isolation modularity. When you are developing multi-tiered applications, multiple users will be performing concurrently interleaved transactions that are on the same database tables. A database driver has to employ sophisticated locking and data buffering algorithms and mechanisms to implement the transaction isolation required for a large-scale JDBC application. This is more complex when there are multiple Java objects working on many databases that could be scattered across the globe. Only time will tell what special needs for transaction isolation there will be in the new Internet/intranet paradigm.

After you have a successful Connection object to a data source, you can interact with the datasource in many ways. The most common approach, from an application developer standpoint, is using the objects that handle the SQL statements. In JDBC, there are three main types of statements:

- Statement
- PreparedStatement
- CallableStatement

The Connection object has the createStatement(), prepareStatement(), and prepareCall()methods to create these statement objects. Chapter , “JDBC Explored,” deals with the statement-type objects in detail. Another notable method in the Connection object is the getMetadata() method that returns an object of the DatabaseMetaData type, which is the topic for the following section.

10.4 METADATA FUNCTIONS

Speaking theoretically, metadata is information about data. The MetaData methods are mainly aimed at the database tools and wizards that need information about the capabilities and structure of the underlying DBMS. Many times these tools need dynamic information about the resultset, which a SQL statement returns. JDBC has two classes of metadata: ResultSetMetaData and DatabaseMetadata. As you can see from the method tables, a huge number of methods are available in this class of objects.

DatabaseMetaData DatabaseMetaDatas are similar to the catalog functions in ODBC, where an application queries the underlying DBMS’s system tables and gets information. ODBC returns the information as a resultset. JDBC returns the results as a ResultSet object with well-defined columns. The DatabaseMetaData object and its methods give a lot of information about the underlying database. This information is more useful for database tools, automatic data conversion, and gateway programs. Table 10.4 gives all the methods for the DatabaseMetaData object. As youcan see, it is a very long table with more than 100 methods. Unless they are very exhaustive GUI tools, most of the programs will not use all the methods. But, as a developer, there will be times when one needs to know some characteristic about the database or to see whether a feature is supported. It is those times when the following table comes in handy.

Table 10.4 DatabaseMetaData Methods

Return Type	**Method Name**	**Parameter**
boolean	allProceduresAreCallable	()
boolean	allTablesAreSelectable	()
String	getURL	()
String	getUserName	()
boolean	isReadOnly	()
Metadata Functions	boolean nullsAreSortedHigh	()
boolean	nullsAreSortedLow	()
boolean	nullsAreSortedAtStart	()
boolean	nullsAreSortedAtEnd	()
String	getDatabaseProductName	()
String	getDatabaseProductVersion	()
String	getDriverName	()
String	getDriverVersion	()
int	getDriverMajorVersion	()
int	getDriverMinorVersion	()
boolean	usesLocalFiles	()
boolean	usesLocalFilePerTable	()
boolean	supportsMixedCaseIdentifiers	()
boolean	storesUpperCaseIdentifiers	()
boolean	storesLowerCaseIdentifiers	()
boolean	storesMixedCaseIdentifiers	()
boolean	supportsMixedCaseQuotedIdentifiers	()
boolean	storesUpperCaseQuotedIdentifiers	()
boolean	storesLowerCaseQuotedIdentifiers	()
boolean	storesMixedCaseQuotedIdentifiers	()
String	getIdentifierQuoteString	()
String	getSQLKeywords	()
String	getNumericFunctions	()
String	getStringFunctions	()
String	getSystemFunctions	()
String	getTimeDateFunctions	()
String	getSearchStringEscape	()
String	getExtraNameCharacters	()
boolean	supportsAlterTableWithAddColumn	()
supportsAlterTable	WithDropColumn	()
boolean	supportsColumnAliasing	()
boolean	nullPlusNonNullIsNull	()
boolean	supportsConvert	()
boolean	supportsConvert	(int
fromType,	inttoType)	
boolean	supportsTableCorrelationNames	()
boolean	supportsDifferentTableCorrelation	()

Names

boolean	supportsExpressionsInOrderBy	()
boolean	supportsOrderByUnrelated	()
boolean	supportsGroupBy	()
boolean	supportsGroupByUnrelated	()
boolean	supportsGroupByBeyondSelect	()
boolean	supportsLikeEscapeClause	()
boolean	supportsMultipleResultSets	()
boolean	supportsMultipleTransactions	()
boolean	supportsNonNullableColumns	()
boolean	supportsMinimumSQLGrammar	()
boolean	supportsCoreSQLGrammar	()
boolean	supportsExtendedSQLGrammar	()
boolean	supportsANSI92EntryLevelSQL	()
boolean	supportsANSI92IntermediateSQL	()
boolean	supportsANSI92FullSQL	()
boolean	supportsIntegrityEnhancement	()

Facility

boolean	supportsOuterJoins	()
boolean	supportsFullOuterJoins	()
boolean	supportsLimitedOuterJoins	()
String	getSchemaTerm	()
String	getProcedureTerm	()
String	getCatalogTerm	()
boolean	isCatalogAtStart	()
String	getCatalogSeparator	()
boolean	supportsSchemasInDataManipulation	()
boolean	supportsSchemasInProcedureCalls	()
boolean	supportsSchemasInTableDefinitions	()
boolean	supportsSchemasInIndexDefinitions	()
boolean	supportsSchemasInPrivilege	()

Definitions

boolean	supportsCatalogsInDataManipulation	()
boolean	supportsCatalogsInProcedureCalls	()
boolean	supportsCatalogsInTableDefinitions	()
boolean	supportsCatalogsInIndexDefinitions	()
boolean	supportsCatalogsInPrivilege	()

Definitions

boolean	supportsPositionedDelete	()
boolean	supportsPositionedUpdate	()
boolean	supportsSelectForUpdate	()
boolean	supportsStoredProcedures	()
boolean	supportsSubqueriesInComparisons	()
boolean	supportsSubqueriesInExists	()
boolean	supportsSubqueriesInIns	()
boolean	supportsSubqueriesInQuantifieds	()
boolean	supportsCorrelatedSubqueries	()
boolean	supportsUnion	()
boolean	supportsUnionAll	()
boolean	supportsOpenCursorsAcrossCommit	()

boolean	supportsOpenCursorsAcrossRollback	()
boolean	supportsOpenStatementsAcrossCommit	()
boolean	supportsOpenStatementsAcrossRollback	()
int	getMaxBinaryLiteralLength	()
int	getMaxCharLiteralLength	()
int	getMaxColumnNameLength	()
int	getMaxColumnsInGroupBy	()
int	getMaxColumnsInIndex	()
int	getMaxColumnsInOrderBy	()
int	getMaxColumnsInSelect	()
int	getMaxColumnsInTable	()
int	getMaxConnections	()
int	getMaxCursorNameLength	()
int	getMaxIndexLength	()
int	getMaxSchemaNameLength	()
int	getMaxProcedureNameLength	()
int	getMaxCatalogNameLength	()
int	getMaxRowSize	()
boolean	doesMaxRowSizeIncludeBlobs	()
int	getMaxStatementLength	()
int	getMaxStatements	()
int	getMaxTableNameLength	()
int	getMaxTablesInSelect	()
int	getMaxUserNameLength	()
int	getDefaultTransactionIsolation	()
boolean	supportsTransactions	()
boolean	supportsTransactionIsolationLevel	(int level)
boolean	supportsDataDefinitionAndDataManipulationTransactions	()
boolean	supportsDataManipulationTransactionsOnly	()
boolean	dataDefinitionCausesTransactionCommit	()
boolean	dataDefinitionIgnoredInTransactions	()

Return Type Method Name Parameter

Metadata Functions

ResultSet	getProcedures	(String catalog, String schemaPattern, String procedureNamePattern)
ResultSet	getProcedure	Columns (String catalog, String schemaPattern, String procedureNamePattern, String columnNamePattern)
ResultSet	getTables	(String catalog, String schemaPattern, String tableNamePattern, String types[])
ResultSet	getSchemas	()
ResultSet	getCatalogs	()
ResultSet	getTableTypes	()

ResultSet	getColumns	(String catalog, Strings chemaPattern, String tableNamePattern, String columnNamePattern)
ResultSet	getColumn Privileges	(String catalog, String schema, String table,Stringn columnNamePattern)
ResultSet	getTablePrivileges	(String catalog, String schemaPattern, StringtableNamePattern)
ResultSet	getBestRowIdentifier	(String catalog, String bschema, String table,int scope, booleannullable)
ResultSet	getVersionColumns	(String catalog, String schema, String table)
ResultSet	getPrimaryKeys	(String catalog, String schema, String table)
ResultSet	getImportedKeys	(String catalog, Stringschema, String table)
ResultSet	getExportedKeys	(String catalog, String schema, String table)
ResultSet	getCrossReference	(String primaryCatalog, String primarySchema, String primaryTable,String foreignCatalog, String foreignSchema, String foreignTable)
ResultSet	getTypeInfo	()
ResultSet	getIndexInfo	(String catalog, String schema, String table, boolean unique, boolean approximate)

As you can see in the table, the DatabaseMetaData object gives information about the functionality and limitation of the underlying DBMS. An important set of information that is very useful for an application programmer includes the methods describing schema details of the tables in the database, as well as table names, stored procedure names, and so on. An example of using the DatabaseMetaData objects from a Java application is the development of multi-tier, scalable applications. A Java application can query if the underlying database engine supports a particular feature. If it does not, Java can call alternative methods to perform the task. This way, the application will not fail if a feature is not available in the DBMS. At the same time, the application will exploit advanced functionality whenever it is available. This is what some experts call “interoperable and yet scalable.” Interoperability is needed for application tools also—especially for general-purpose design and query tools based on Java that must interact with different data sources. These tools have to query the data source system to find out the supported features and proceed accordingly. The tools might be able to process information faster with data sources that support advanced features, or they may be able to provide the user with more options for a feature-rich data source.

ResultSetMetaData:

Compared to the DatabaseMetaData, the ResultSetMetaData object is simpler and has fewer methods. But these will be more popular with application developers. The ResultSetMetaData, as the name implies, describes a ResultSet object. Table 10.5 lists all the methods available for the ResultSetMetaData object. Return Type Method Name ParameterMetadata Functions

Table 10.5 ResultSetMetaData Methods

Int	getColumnCount	()
boolean	isAutoIncrement	(int column)
boolean	isCaseSensitive	(int column)
boolean	isSearchable	(int column)
boolean	isCurrency	(int column)
int	isNullable	(int column)
boolean	isSigned	(int column)
int	getColumnDisplaySize	(int column)
String	getColumnLabel	(int column)
String	getColumnName	(int column)
String	getSchemaName	(int column)
int	getPrecision	(int column)
int	getScale	(int column)
String	getTableName	(int column)
String	getCatalogName	(int column)
int	getColumnType	(int column)
String	getColumnTypeName (int column)	
boolean	isReadOnly	(int column)
boolean	isWritable	(int column)
boolean	isDefinitelyWritable	(int column)

Return Values

int columnNo

Nulls = 0

int column

Nullable = 1

int Column

Nullable

Unknown = 2

As you can see from the preceding table, the ResultSetMetaData object can be used to find out about the types and properties of the columns in a resultset. You need to use methods such as getColumnLabel() and getColumnDisplaySize() even in normal application programs. Using these methods will result in programs that handle result sets generically, thus assuring uniformity across various applications in an organization as the names and sizes are taken from the database itself. Before you leave this chapter, also look at the exception handling facilities offered by JDBC.

10.5 THE SQLEXCEPTION CLASS

The SQLException class in JDBC provides a variety of information regarding errors that occurred during a database access. The SQLException objects are chained, so a program can read them in order. This is a good mechanism, as an error condition can generate multiple errors and the final error might not have anything to do with the actual error condition. By errormessage and vendor-specific error code. Also associated with a SQLException is a SQLStatestring that follows the XOPEN SQLState values defined in the SQL specification. Table 10.6 list sthe methods for the SQLException class.
Table 10.6 SQLException Methods

Return Type   Method Name   Parameter

SQLException SQLException (String reason, String SQLState, intvendorCode)

SQLException SQLException (String reason, String SQLState)

SQLException SQLException (String reason)

SQLException SQLException ()String

getSQLState   ()

int   getErrorCode ()

SQLException getNextException ()

void   setNextException (SQLException ex)

10.6 THE SQLWARNINGS CLASS

Unlike the SQLException class, the SQLWarnings class does not cause any commotion in a Java program. The SQLWarnings are tagged to the object whose method caused the warning. So you should check for warnings using the getWarnings() method that is available for all objects.

Table 10.7 lists the methods associated with the SQLWarnings class.

The SQLWarnings Class

Table 10.7 SQLWarnings Methods

Return Type Function Name Parameter

SQLWarning SQLWarning (String reason, String SQLState, int vendorCode)

SQLWarning SQLWarning (String reason, String SQLState)

SQLWarning SQLWarning (String reason)

SQLWarning SQLWarning ()
SQLWarning getNextWarning ()
void setNextWarning (SQLWarning

10.7 SUMMARY

In this chapter we covered overview of JDBC. How JDBC works. The applet and the JDBC layers communicate in the client system, and the driver takes care of interacting with the database over the network. JDBC class hierarchy and a JDBC API flow. We have covered JDBC example database schema. ODBC Setup for the example database. In JDBC, there are three main types of statements: Statement, PreparedStatement, CallableStatement. We have covered two classes of metadata: ResultSetMetaData and DatabaseMetadata.

10.8 QUESTIONS

1. Explain how JDBC works.
2. Explain different Statements of JDBC.
3. Explain different functions of MetaData.

*****

11

JAVA BEANS

Unit Structure

11.1 SELF-CONTAINED COMPONENTS

JavaBeans adds to the Java platform the capability to create a complete application by simply linking together a set of self-contained components. Microsoft's Visual Basic and Borland's Delphi are both examples of applications that allow users to build full-blown applications by combining independent components. The success and popularity of these two applications alone speak volumes to the success of this style of application building. Just as with other models, there is no restriction on the size or complexity of a JavaBeans component. In principle, JavaBeans components (or just Beans) can range from widgets and controls to containers and applications. The philosophy behind JavaBeans is to provide easy-to implement functionality for the former, while allowing enough flexibility for the latter. In the spirit of this philosophy, you'll see how to create and use fairly simple Beans. However, after you finish reading this chapter, you'll have learned enough to create larger and more complex Beans, if you choose to do so. If you want to dive deeper into all the intricacies of JavaBeans, you might want to purchase a copy of Que's *Special Edition Using JavaBeans*, which goes into more detail on all the topics discussed in this chapter.

11.2 IMPORTANT CONCEPTS IN COMPONENT MODELS

JavaBeans provides a platform-neutral component architecture. Examples of non-platform–neutral component architectures include COM/OLE for the Windows platform and OpenDoc for the Macintosh platform. A component written to be placed into an OpenDoc container, like ClarisWorks for example, can’t be used inside a COM/OLE container like Microsoft Word. Because JavaBeans is architecture-neutral, Beans can be placed into any container for which abridge exists between JavaBeans and the container’s component architecture. Thus, a JavaBeans component could be used in both Microsoft Word and ClarisWorks. To accomplish this seemingly impossible feat, the JavaBeans specification adopts features common with the other popular component models. In particular, these features include the following:

- Component fields or properties
- Component methods or functions
- Events and intercommunication
- State persistence and storage

If you are familiar with component models already, you don’t necessarily need to read this section. You can jump right into the next section, “The Basics of Designing a Java Bean.”

Component Fields or Properties:

For a component to be useful, it has to have a set of properties that define its state. For example, if you were to design a component that displayed some text, one of the properties of that component might be the foreground color of the font. Another property might be the type and size of the font. Taken as a whole, the set of properties that make up a component also define its state. For example, if the properties of one component completely match that of another, they are in the same state. Properties are often used to define not only the appearance but also the behavior of components. This is because a component need not have any appearance at all. For example, a component in a spreadsheet might calculate the interest earned on some earnings column. If that component is not capable of displaying the data, then it probably shouldn’t have any properties associated with appearance. It is likely, however, that it will have a property that defines the current interest rate. Properties can range from Boolean values, to strings, to arrays, to other components. They can also be interdependent. Following the same example above, a component that displays the earnings column might want to be notified if the interest rate property of the other component changes.

Component Methods or Functions:

The API, so to speak, of a component is the collection of methods or functions that it contains that other components and containers can call. There has to be some way for a container to modify a component’s properties, notify it of an event (see below), or execute some functionality. Different component models differ in how they make the properties and methods of their components available to other components. Because entire books have been written on how this is implemented for different models, suffice it to say that this is a common feature of component models. This topic will be discussed as it relates to JavaBeans in the section on Introspection later in the chapter.

Events and Intercommunication:

A component by itself is a lonely component. Even though some components might have extensive functionality and many properties, in the true spirit of a component, it should only be useful when used in conjunction with other components. So if two components are sitting together in a container, how do they talk? How does one let the other know when it has done something the other really ought to know about?

The method by which most components communicate is through *event transmission*. One component (or the container) undergoes some action causing it to generate an event. For example, an event is generated when you click a button. Depending on the model, the component will notify the container, the interested components, or both, of the events. At the sametime, the objects in the environment also act on events delivered to them. For example, the Filedialog box displays itself when it hears that you just clicked a Browse button.

State Persistence and Storage:

It is important for components to remember their state. This is so common that you may not even recognize it. When you open an application and it remembers the size and position of its window when it was last closed, it is maintaining (to some degree) a persistent state.

Important Concepts in Component Models:

Also important is the capability to store and retrieve components. Sun Microsystems, Inc. likes to call this *packaging*. This is especially important in a distributed environment where the components are likely to be served up over a network.

11.3 THE BASICS OF DESIGNING A JAVA BEAN

All good programmers recognize the importance of the design phase in programming. Thus, you’ll start out by addressing how to design a Bean. As you will learn later, the way in which you design your Bean directly affects the way it behaves in containers. For example, the names you choose for the methods should follow specific design specifications. If you start from the beginning with these rules in mind, allowing your Bean to participate in Introspection does not require any additional programming on your part. Don’t worry so much right now about what Introspection is; you’ll get into that later.

Designing a Bean consists of the following steps:

1. Specifying the Bean’s properties
2. Specifying the events the Bean generates or responds to
3. Defining which properties, methods, and events the Bean exposes to other Beans or to its container
4. Deciding miscellaneous issues, such as whether the Bean has its own Customization dialog box or whether it requires some prototypical state information

You’ll start by designing some Beans. For the sake of a simple example, assume that you are developing two Beans; one Bean allows text to be entered into it, and the other displays some text. You can imagine how these Beans might be useful. By placing these two Beans into a container, you can use one to enter text that the other will then display. What types of properties do you think these Beans need to have? What events are these Beans interested in hearing about? What events do these Beans generate? Do these Beans expose all of their properties and events, or just some? At this point, you may not know the answers to these questions. The process is the important concept here; the details will become clearer as you progress through the chapter. Regardless, the first thing any Bean needs is a name. In this chapter, the sample Beans will be called TextDisplayer and TextReader.

Specifying the Bean’s Properties:

The TextDisplayer and TextReader Beans definitely need to have a property defining the text they hold. For example, say that the TextDisplayer Bean also contains properties defining the background color, the font, and the font color. The TextReader Bean also contains a property that defines how many columns of characters it can display. Table 11.1 lists the TextDisplayer Bean’s properties and the Java types that will be used to implement them. Table 11.2 lists the TextReader Bean’s properties and the Java types that will be used for them.

Table 11.1 The TextDisplayer Bean’s Properties and Java Types

Property Name	Java Type
OutputText	java.lang.String

BGColor	java.awt.Color
TextFont	java.awt.Font
FontColor	java.awt.Color

Table 11.2 The TextReader Bean's Properties and Java Types

Property Name	Java Type
InputText	java.lang.String
Width	int

Specifying the Events the Bean Generates or Responds To

Our TextDisplayer Bean must respond to an event specifying that its text should change. Specifically, it must update its OutputText property and redraw itself. The TextReader Bean doesn't need to respond to any events, but it must generate (or fire) an event when the user changes its InputText property. This type of event is called a PropertyChangeEvent, for obvious reasons.

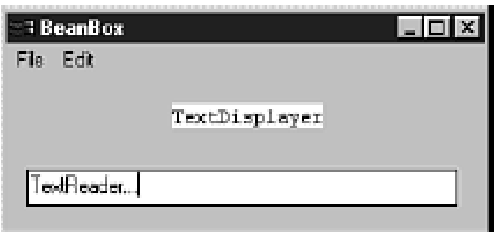
Properties, Methods, and Event Exposure:

Because these Beans are particularly simple, you don't need to hide anything from the Beans' container or the other Beans interested in them. JavaBeans provides a mechanism for you that will use the names of your methods to extract the names and types of your properties and events. Rest assured that you will learn how this works as you go along. Later in the chapter, you'll learn how to explicitly define what information in a Bean is exposed to its environment.

Initial Property Values and Bean Customizers:

You want to keep this example simple, so assume that your Beans do not need any prototypical information (you'll define default values for all their properties) and that they do not have their own Customization dialog box. This means that your Beans have a predefined state when they're instantiated and that they use the standard PropertyEditors for their properties. If you were designing a Bean that displays an HTML page, for example, specifying default values might not be possible. You would need to know what file to display when the Bean is instantiated. At this point, you've designed your Beans enough to begin coding. This will be an additive process because you haven't learned how to make the Beans do anything yet. All the code required to actually display the Beans isn't included in Listings 11.1 and 11.2 because it'smainly AWT-related and isn't relevant to this chapter. If you want to see the entire listings, please refer to the CD-ROM. In Figure 11.1 you can see your Beans hard at work inside the BeanBox. The BeanBox is a JavaBeans container that you can download from Sun's Web site; it's included in the BDK, or Beans Development Kit. Right now, the Beans are completely isolated. Because you haven't given the Beans any functionality yet, this is about as good as it gets. The preliminary code needed to instantiate our TextDisplayer Bean is shown in Listing11.1.

FIG. 11.1 Sun's BeanBox showing the TextDisplayer and TextReader Beans.



Listing 11.1 TextDisplayer.java—Preliminary Code for the TextDisplayer
Bean

```
public class TextDisplayer extends Canvas implements
PropertyChangeListener {
// default constructor for this Bean. This is the constructor that
an
// application builder (like Visual Basic) would use.
public TextDisplayer() {
this( “TextDisplayer”, Color.white, new Font( “Courier”,
Font.PLAIN, 12 ),
Color.black );
}
// custom constructor for this Bean. This is the constructor you
would
// likely use if you were going to do all your coding from scratch.
public TextDisplayer( String OutputText, Color BGColor, Font
TextFont,
Color FontColor ) {
super(); // call the Canvas’s constructor.
this.OutputText = OutputText;
this.BGColor = BGColor;
this.TextFont = TextFont;
this.FontColor = FontColor;
setFont( TextFont ); // set the Canvas’s font.
setBackground( BGColor ); // set the Canvas’s background
color.
setForeground( FontColor ); // set the Canvas’s foreground
color.
}
// this Bean’s properties.
protected String OutputText;
protected Color BGColor, FontColor;
protected Font TextFont;
}
```

You might have noticed you have specified that your Bean implement an interface called PropertyChangeListener. This is so that the TextDisplayer Bean can update its OutputText property by receiving an event. How that works will be discussed in more detail later in the chapter. The preliminary code needed to instantiate your TextReader Bean is shown in Listing 11.2.

Listing 11.2 TextReader.java—Preliminary Code for the TextReader Bean

```
Public class textreader extends textfield {
// default constructor for this bean. This is the constructor that an
// application builder (like visual basic) would use.
Public textreader() {
This( “”, 40 );
}
// custom constructor for this bean. This is the constructor that you would
// likely use if you were doing your coding from scratch.
Public textreader( string inputtext, int width ) {
Super( inputtext, width );
This.inputtext = inputtext;
This.width = width;
setEditable( true );
}
// this Bean’s properties.
protected String InputText;
protected int Width;
}
```

The Basics of Designing a JavaBean :

Creating and Using Properties

In Figure 11.1, you will notice that the TextDisplayer Bean displayed itself with a white background and black text. It did so because that’s how you set its properties. If you had set the FontColor property to red, it would have displayed the text in red. If the properties of a component cannot be changed by other Beans, the usefulness of the Bean is reduced, as well as the reusability. For example, if you used the TextDisplayer Bean in an accounting package, you would need to change the Bean’s FontColor property to red to indicate a negative value. So how do you let other Beans know that they can set (or read) this property? If you’re coding from scratch, you can look at the documentation for the Bean. But what if you’re in an application builder? Luckily, there's a way to do this without incurring any extra coding on your part.

You'll see how that works a little later.

Two types of properties are supported by JavaBeans: single-value and indexed. In addition, properties can also be bound or constrained. A single-value property is a property for which there is only one value. As the name suggests, an indexed property has several values, each of which has a unique index. If a property is bound, it means that some other Bean is dependent on that property. In the continuing example, the TextReader Bean's InputText property is bound to our TextDisplayer Bean; the TextReader must notify the TextReader after its InputText field changes. A property is constrained if it must check with other components before it can change. Note that constrained properties cannot change arbitrarily—one or more components may not allow the updated value.

Single-Value Properties :

All properties are accessed by calling methods on the owning Bean's object. Readable properties have a getter method used to read the value of the property. Writable properties have a setter method used to change the value of a property. These methods are not constrained to simply returning the value of the property; they can also perform calculations and return someother value. All the properties our Beans have are single-value. At this point, you're ready to start talking about Introspection. The method by which other components learn of your Bean's properties depends on a few things. In general, though, this process is called Introspection. In fact, the class java.beans.Introspector is the class that provides this information for other components. The Introspector class traverses the class hierarchy of a particular Bean. If it finds explicit information provided by the Bean, it uses that. However, it uses design patterns to implicitly extract information from those Beans that do not provide information. Note that this is what happens for your Beans. Specific design rules should be applied when defining accessor methods so that the Introspector class can do its job. If you choose to use other names, you can still expose a Bean's properties, but it requires you to supply a BeanInfo class. For more about what a BeanInfo class is, see the section on Introspection. Here are the design patterns you should use:

```
public void set<PropertyName>( <PropertyType> value );
public <PropertyType> get<PropertyName>();
public boolean is<PropertyName>();
```

Note that the last pattern is an alternative getter method for Boolean properties only. Setter methods are allowed to throw exceptions if they so choose. The accessor methods for the TextDisplayer Bean are shown in Listing 11.3. Notice that all the accessor methods have been declared as synchronized. Even though nothing serious could happen in this Bean, you should always assume that your Beans are running in multithreaded environments. Using synchronized accessor methods helps prevent race conditions from forming. You can check theTextReader.java file on your CD-ROM to see the accessor methods for the TextReader Bean.

Listing 11.3 TEXTDISPLAYER.JAVA—The Accessor Methods for the Properties in the TextDisplayer Bean

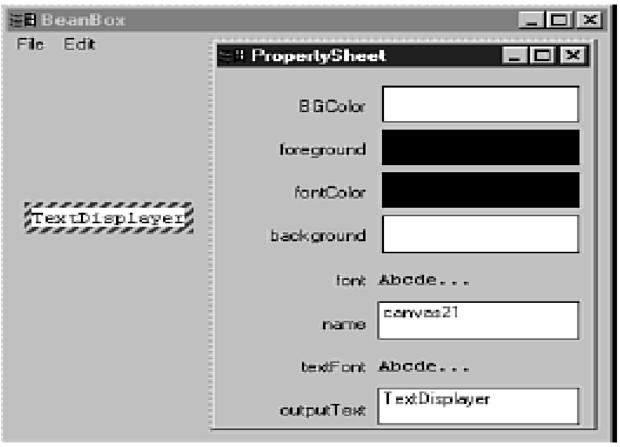
```
public synchronized String getOutputText() {
return( OutputText );
}
public synchronized void setOutputText( String text ) {
OutputText = text;
resizeCanvas();
}
public synchronized Color getBGColor() {
return( BGColor );
}
public synchronized void setBGColor( Color color ) {
BGColor = color;
setBackground( BGColor ); // set the Canvas’s background color.
repaint();
}
public synchronized Font getTextFont() {
return( TextFont );
}
public synchronized void setTextFont( Font font ) {
TextFont = font;
setFont( TextFont ); // set the Canvas’s font.
resizeCanvas();
}
public synchronized Color getFontColor() {
return( FontColor );
}
public synchronized void setFontColor( Color color ) {
FontColor = color;
setForeground( FontColor ); // set the Canvas’s foreground color.
repaint();
}
```

11.4 CREATING AND USING PROPERTIES

Figure 11.2 shows you what the property sheet of Sun's BeanBox shows for your TextDisplayer Bean. Notice that you can see the properties of the parent class, too. Your Bean inherits from java.awt.Canvas, which inherits from java.awt.Component, which inherits from java.lang.Object. The additional properties that you see are from the java.awt.Component class. This illustrates the principal drawback of using the automatic JavaBeans Introspection methods. In your own Beans, this might be the motivation for providing a BeanInfo class. Again, more on that is in the section on Introspection.

FIG. 11.2

The PropertySheet of Sun's BeanBox showing the Bean's exposed properties. Notice the properties of the parent class.



font Abcde...

name canvas21

textFont Abcde...

outputText TextDisplayer

Indexed Properties:

All indexed properties must be Java integers. Indexed properties can be read individually or as an entire array. The design patterns for indexed properties are as follows:

public <PropertyType> get<PropertyName>(int index);

public void set<PropertyName>(int index, <PropertyType> value);

public <PropertyType>[] get<PropertyName>();

public void set<PropertyName>(<PropertyType>[] value);

To illustrate, assume there is a Meal property that consists of an array of Courses:

public Course getMeal(int course);

public void setMeal(int course, Course dish);

public Course[] getMeal();

public void setMeal(Course[] dishes);

Bound Properties :

As the programmer, you can decide which of your Bean’s properties other components can bind to. To provide bound properties in your Beans, you must define the following methods:

```
public voidaddPropertyChangeListener(
PropertyChangeListener l );
public void removePropertyChangeListener(
PropertyChangeListener l );
```

To provide this functionality on a per-property basis, the following design pattern should be used:

```
public void add<PropertyName>Listener(
PropertyChangeListener l );
public void remove<PropertyName>Listener(
PropertyChangeListener l );
```

Beans wanting to bind to other components’ properties should implement thePropertyChangeListener interface, which consists of the following method:

```
public void propertyChange( PropertyChangeEvent evt );
```

Whenever a bound property in a Bean is updated, it must call the propertyChange() method in all the components that have registered with it. The class java.beans.PropertyChangeSupport is provided to help you with this process. The code in Listing 11.4 shows you what is required in the TextReader Bean to allow its InputText property to be bound.

Listing 11.4. TEXTREADER.JAVA—Code Required to Make the InputText

Property of the TextReader Bean a Bound Property

```
// setter method for the InputText property.
public synchronized void setInputText( String newText ) {
String oldText = InputText;
InputText = newText;
setText( InputText );
changeAgent.firePropertyChange( “inputText”, new String(
oldText ),
new String( newText ) );
}
// these two methods allow this Bean to have bound properties.
public void addPropertyChangeListener(
PropertyChangeListener l ) {
changeAgent.addPropertyChangeListener( l );
}
public void removePropertyChangeListener(
PropertyChangeListener l ) {
```

```
changeAgent.removePropertyChangeListener( l );
}
protected PropertyChangeSupport changeAgent = new PropertyChangeSupport( this
);
```

Constrained Properties:

The process for providing constrained properties in your code is also fairly straightforward. You must define the following methods in your Bean:

```
public void addVetoableChangeListener( VetoableChangeListener l );
public void removeVetoableChangeListener( VetoableChangeListener l );
```

Just as with bound properties, you can make individual properties constrained using the followingdesign pattern:

```
public void add<PropertyName>Listener( VetoableChangeListener l );
public void remove<PropertyName>Listener( VetoableChangeListener l );
```

Beans intended to constrain other components' properties should implement theVetoableChangeListener interface, which consists of the following method:

```
public void vetoableChange( PropertyChangeEvent evt );
```

Whenever a constrained property in a Bean is updated, it must call the vetoableChange()method in all the components that have registered with it. There is also a support class to helpmake this process easier. Use the class java.beans.VetoableChangeSupport to help manageyour vetoable properties. The code in Listing 11.5 shows you what is required in theTextReader Bean to allow its Width property to be constrained.

Listing 11.5 TEXTREADER.JAVA—Code Required to Make the Columns

Property of the TextReader Bean a Constrained Property

```
// setter method for the Columns property.
public synchronized void setWidth( int newWidth )
throws PropertyVetoException {
int oldWidth = Width;
vetoAgent.fireVetoableChange( “width”, new Integer( oldWidth ),
new Integer( newWidth ) );
// no one vetoed, so change the property.
Width = newWidth;
setColumns( Width );
Component p = getParent();
if ( p != null ) {
p.invalidate();
p.layout();
}
changeAgent.firePropertyChange( “width”, new Integer(
oldWidth ),
new Integer( newWidth ) );
}
// these two methods allow this Bean to have constrained
properties.
public void addVetoableChangeListener(
VetoableChangeListener l ) {
vetoAgent.addVetoableChangeListener( l );
}
public void removeVetoableChangeListener(
VetoableChangeListener l ) {
vetoAgent.removeVetoableChangeListener( l );
}
protected VetoableChangeSupport vetoAgent = new
VetoableChangeSupport( this
);
```

In this particular example, we chose to make the Width property bound and constrained. A property does not have to be bound to be constrained. For example, to make the Width property constrained but not bound, we would remove the following line from Listing 11.5:

```
        changeAgent.firePropertyChange( “width”, new Integer(
oldWidth ),
new Integer( newWidth ) );
```

11.5 USING EVENTS TO COMMUNICATE WITH OTHER COMPONENTS

The whole idea behind the JavaBeans component model is to provide a way to create reusable components. To do this, Beans must be able to communicate with the other Beans in their environment and with their container. This is accomplished by means of Listener interfaces. You’ve already seen some of this with the PropertyChangedEvent from the last section. More detail about how this works follows. Beans use the same event-handling scheme as AWT. This means that if your Bean needs to hear about events coming from another Bean, it must register itself with that Bean. To do this, it must implement the Listener interface for the event of interest. At the same time, if your Bean is no longer interested in hearing about some other Bean’s event, it must unregister itself with that Bean. Any event that a Bean wants to fire must inherit from the java.util.EventObject class. For simple events, the java.util.EventObject class itself could be used; however, as with java.lang.Exception, using child classes provides clarity andis preferred. All Listener interfaces must inherit from the java.util.EventListener interface,and the same subclassing convention applies. The event handling method of a Listenerinterface should follow the design pattern for Introspection as shown here:

void <EventOccuranceName>(<EventObjectType evt);

Note that <EventObjectType> must inherit from java.util.EventObject. Here is an example of an event handler for a DinnerServedListener interface:

void dinnerServed(DinnerServedEvent evt); // DinnerServedEvent inherits from

// java.util.EventObject.

There is no restriction preventing an event handler method from throwing an exception. In addition, any one Listener interface can have any number of related event handlers. There are two types of events that components can listen for: multicast events and unicastevents.

1.Multicast Events:

Multicast events are the most common types of events. The PropertyChangeEvent, which you have already been exposed to, is a multicast event because there can be any number of listeners. In that example, you had addPropertyChangeListener() and removePropertyChangeListener() methods, which allowed other components to register with the Bean as being interested in hearing when a bound property changed. The process is the same for any other type of multicast event, and the registration methods should follow the design pattern for Introspection as shown here:

public synchronized void add<ListenerType>(<ListenerType> listener);
public synchronized void remove<ListenerType>(<ListenerType> listener);

Using Events to Communicate with Other Components
The keyword synchronized is not actually part of the design pattern. It is included as a reminder that race conditions can occur, especially with the event model, and precautions must be taken.

2.Unicast Events:

Unicast events don’t occur nearly as often as their counterpart, but they’re just as useful.Unicast events can have only one listener. If additional components attempt to listen to the unicast event, a java.util.TooManyListenersException will be thrown. The following design pattern should be used when declaring unicast events:

public synchronized void add<ListenerType>(<ListenerType> listener) throws

java.util.TooManyListenersException;

public synchronized void remove<ListenerType>(<ListenerType> listener);

Event Adapters:

In some cases, it may be necessary to build an event adapter class that can transfer an event to a component. This comes into play especially for an application builder because the application doesn't know until runtime how the components will be linked together or how they will interact with each other's events.

An event adapter intervenes in the normal event-handling scheme by intercepting the events normally meant for another component. For example, assume that a user places a button and a text box in an application builder. If the user wants the text box to fill with the word "Pressed" when the button is pressed, the application builder can use an event adapter to call a method containing the user-generated code needed to do it. Here's how it will eventually work:

1. The event adapter registers with the event source. In other words, it calls an addSomeEventListener() method on the event source component.

2. The event source component fires an event by calling the event adapter's event-handler method, someEvent(). Keep in mind that the event source component doesn't care whether it's calling an event adapter. At this point, with the event fired, it can continue onwith its business.

3. The event adapter calls the specific user-designed method on the final target component.

4. The code in the user-designed method fills in the text box component with the "Pressed" text. Sometimes it helps to see some code. Listing 11.6 contains some pseudocode you can examine to see how an event adapter is written. The code in the example builds off the procedure listed previously. You won't be able to compile this code (notice the class keywords have beenchanged to pseudoclass), but it serves as an example you can build off of in your own Beans.

Listing 11.6. ADAPTOREXAMPLE.JAVA—Pseudocode Showing How to

Implement an Adapter Class; This Code Might Be Generated by an Application Builder

```
// this pseudoclass example uses a unicast mechanism to keep things simple.
public interface SomeEventListener extends java.util.EventListener {
public someEvent( java.util.EventObject e );
}
public pseudoclass button extends java.awt.Button {
public void synchronized addSomeEventListener( SomeEventListener l )
throws java.util.TooManyListenersException {
if ( listener != null ) {
listener = l;
} else throw new java.util.TooManyListenersException;
}
private void fireSomeEvent() {
listener.someEvent( new java.util.EventObject( this ) );
}
private SomeEventListener listener = null;
}
public pseudoclass eventAdaptor implements SomeEventListener {
public eventAdaptor( TargetObject target ) {
this.target = target;
}
someEvent( java.util.EventObject e ) {
// transfer the event to the user generated method.
target.userDefinedMethod();
}
private TargetObject target;
}
public pseudoclass TargetObject {
public TargetObject() {
adaptor = new eventAdaptor( this );
}
public userDefinedMethod() {
// user generated code goes here.
}
private eventAdaptor adaptor;
}
```

11.6 INTROSPECTION:CREATING AND USING BEANINFO CLASSES

You’ve already seen in the preceding sections and in the two Beans you designed how to use design patterns to facilitate automatic Introspection. You also saw that the automatic Introspection mechanism isn’t perfect. If you look back at Figure 11.2, you’ll see an example of this. Introspection is probably the most important aspect of JavaBeans because without it a container can’t do anything with a Bean other than display it. As you become proficient at designing your own Beans, you’ll find that you sometimes need to provide additional Introspection information for the users of your Beans. In the case of your Beans, this is to hide the parent class’s properties to clear up ambiguities. The java.beans.Introspector class, as discussed earlier in the chapter, does all the pattern analysis to expose the properties, methods, and events that a component has. As a first step, though, this class looks to see whether a BeanInfo class is defined for the Bean it’s inspecting. If it finds one, it doesn’t do any pattern analysis on the areas of the Bean for which the BeanInfo class supplies information. This means that you can selectively choose which information you want to provide and which information you want to be derived from analysis. Toshow how this is done, you’ll design a BeanInfo class for our TextDisplayer Bean.The first thing you need to do is define what information you’ll provide and what you’ll leave upto the Introspector class to analyze. For the sake of example, say that you’ll choose to provide the properties of your Bean, and you’ll let the Introspector class use analysis to expose the events and methods. Table 11.5 shows the names of the TextDisplayer Bean’s properties and the user-friendly names you want to display. With that information defined, you can start working on your BeanInfo class, TextDisplayerBeanInfo.class. Notice how you simply appended “BeanInfo” to the class name. That’s an Introspection design pattern; the Introspector classlooks for BeanInfo information by appending “BeanInfo” to the class name of the Bean it’scurrently analyzing.

Table 11.5 The TextDisplayer Bean’s Properties and User-Friendly Names

Property Name	User-Friendly Name
OutputText	“Text String”
BGColor	“Background Color”
TextFont	“Text Font”
FontColor	“Text Color”

All BeanInfo classes must implement the java.beans.BeanInfo interface. At first glance, that seems difficult; there are eight methods in the java.beans.BeanInfo interface! But remember the Introspector class has a set procedure for the way it looks for information. For the sake of clarity, that procedure is shown in the following list:

1. The Introspector class looks for a BeanInfo class for the Bean it’s analyzing.
2. If a BeanInfo class is present, each method in the BeanInfo class is called to find outwhether it can provide any information. The Introspector class will use implicit analysisto expose information for which the BeanInfo class denies any knowledge (returns anull value). If no BeanInfo class is found, the Introspector class will use implicitanalysis for all the methods in the java.beans.BeanInfo interface.
3. The Introspector class then checks to see whether it has obtained explicit information for each of the methods in the BeanInfo interface. If it has not, it steps into the parent class (if one exists) and starts the process over for only those methods that it had to use analysis on.
4. When the Introspector class has gotten information from a BeanInfo class for all the methods in the java.beans.BeanInfo interface, or when there are no more parent classes to explore, the Introspector class returns its results. To make your life easier as a programmer, Sun has provided a prebuilt class, xjava.beans.SimpleBeanInfo, that returns a null value for all the BeanInfo methods. That way, you can inherit from that class and override only the methods you choose.

Listing 11.7 shows the BeanInfo class for the TextDisplayer Bean. Notice how you only override the getPropertyDescriptors() method. The parent class returns null for all the other methods in the java.beans.BeanInfo interface.

Listing 11.7 TEXTDISPLAYERBEANINFO.JAVA—The Entire BeanInfo Class for

the TextDisplayer Bean Showing How to Provide Property Information

```
import java.beans.*;
public class TextDisplayerBeanInfo extends SimpleBeanInfo {
// override the getPropertyDescriptors method to provide that info.
public PropertyDescriptor[] getPropertyDescriptors() {
PropertyDescriptor[] properties = new PropertyDescriptor[4];
try {
properties[0] = new PropertyDescriptor(
“Text String”, BeanClass, “getOutputText”,
å”setOutputText” );
properties[1] = new PropertyDescriptor(
“Text Color”, BeanClass,
å”getFontColor”, “setFontColor” );
properties[2] = new PropertyDescriptor(
“Text Font”, BeanClass,
å”getTextFont”, “setTextFont” );
properties[3] = new PropertyDescriptor(
“Background Color”, BeanClass,
å”getBGColor”, “setBGColor” );
} catch( IntrospectionException e ) {
return( null ); // exit gracefully if we get an exception.
}
```

continues

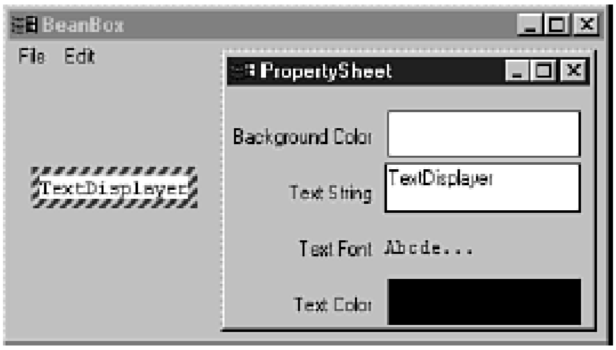
Introspection: Creating and Using BeanInfo Classes

Listing 11.7 Continued

```
return( properties );
}
private Class BeanClass = TextDisplayer.class;
}
```

Take a second to look at the try|catch clause in Listing 11.7. Notice how you return a null value if you catch a java.beans.IntrospectionException. If you catch this exception, it usually means that you’ve provided an incorrect getter or setter method name. You should always return a null value if you catch this exception so that the Introspector class can still analyze your Bean. You should be able to extend this example to override the other methods in the java.beans.BeanInfo interface. Figure 11.3 shows the PropertySheet window of Sun’s BeanBox for our TextDisplayer Bean. Notice how the user-friendly names for the propertieshave been used, and the parent class’s properties are gone. Sweet success!

Fig.11.3 The PropertySheet window of Sun’s BeanBox showing the user-friendly names for the properties in the TextDisplayer Bean.



Background Color

Text String TextDisplayer

Text Font Abcde...

Text Color

11.7 CUSTOMIZATION: PROVIDING CUSTOM PROPERTY EDITORS AND GUI INTERFACES

So far you have seen how to create a Bean; how to expose its properties, methods, and events; and how to tweak the Introspection process. You might have noticed from the figures that the properties of a Bean have a PropertyEditor. For example, look at Figure 11.3. In the PropertySheet window, next to the “Text String” label, there’s a TextField AWT component already filled with the value of the OutputText property. You didn’t supply any code for this component, so how did Sun’s BeanBox know to provide it? The answer is that the BeanBoxapplication asked the java.beans.PropertyEditorManager what the default PropertyEditor was for an object of type java.lang.String, and displayed it. Just because PropertyEditors and Customizers require a GUI environment doesn’t mean aBean can’t function without one. For example, a Bean designed to run on a server might not use (or need) a GUI environment at all. The java.beans.Beans class and the java.beans. Visibility interface allow Beans to have different behavior in GUI and non-GUI environments.

PropertyEditors and the PropertyEditorManager :

The class java.beans.PropertyEditorManager provides default PropertyEditors for the majority of the Java class types. So, if you use only native Java data types and objects, you’re all set. But what if you design a Bean that has a property for which there’s no default PropertyEditor? You’ll run into this problem any time you design a custom property type. For those cases where there is no default PropertyEditor, you have to provide your own. Actually, you could redesign all the default PropertyEditors, too, if you choose, but you would only do this in rare cases, so this won’t be discussed here. This means that you have to provide an additional class, by appending Editor to the class name, that the PropertyEditorManager can use. In most cases, you provide a subclass of java.awt.Component. The property sheet for your component will then pop up your custom PropertyEditor to allow your custom property to be edited. You won’t actually design a custom PropertyEditor here because the majority of Beans won’t require it, but an explanation of how to do it will be included. The requirements of a PropertyEditor are as follows:

1. Custom PropertyEditors must inherit from java.awt.Component so that they can be displayed in a property sheet. Note that this could simply mean inheriting from an AWT component like java.awt.TextField.

2. Custom PropertyEditors must derive their class name by post fixing Editor to the property class name unless they register themselves with the PropertyEditorManager for their container (see step 3). For example, the PropertyEditor for a custom property type CustomProperty.class must be named CustomPropertyEditor.class.

3. For custom PropertyEditors that do not follow the standard naming conventionin step 2, the custom property type must register itself with the container's PropertyEditorManager by calling the registerEditor() method.

4. Custom PropertyEditors must always fire a PropertyChange event to update the custom property. This is a must! Otherwise, the container has no way of knowing to update the component. You might be asking yourself, "Can I provide my own property sheet?" The answer is yes, and for complex Beans, this is absolutely imperative. Property sheets by nature are simple and relatively unuser-friendly. The following section discusses how to override the property sheet mechanism to provide your own customization dialog boxes.

Customization Editor :

All application builders have to implement some method of customizing the Beans placed into their containers. Thus, the PropertyEditor mechanism and the idea of a property sheet were born. But what about the special cases where a Bean can be customized several different ways, or there are dozens of properties? The solution to this problem is called *customizers*. Bean developers can optionally supply customizer classes with their Beans to be used in place of standard property sheets. Even though the property sheet mechanism works just fine for the TextReader Bean, you'll create a customizer class anyway, to learn how it's done.

Customization: Providing Custom PropertyEditors and GUI Interfaces :

To implement a customizer class, a Bean must also provide a BeanInfo class. The class name of a Bean's customizer class is determined from a call to the getBeanDescriptor() method of the java.beans.BeanInfo interface. This is a little bit different from what you've encountered so far. There is no default Introspection design pattern for customizers; you must provide a BeanInfo class, even if the only information it provides is a BeanDescriptor. In fact, this is what you do for the TextReaderBeanInfo.class shown in Listing 11.8. Notice how the class inherits from java.beans.SimpleBeanInfo; the parent class implements the java.beans.BeanInfo class, and you simply override the getBeanDescriptor() method so that it returns something meaningful.

Listing 11.8 TEXTREADERBEANINFO.JAVA—The BeanInfo Class for the

TextReader Bean Showing How to Provide Customizer Class Information

```
import java.beans.*;
public class TextReaderBeanInfo extends SimpleBeanInfo {
// override the getBeanDescriptor method to provide a customizer.
public BeanDescriptor getBeanDescriptor() {
return( new BeanDescriptor( BeanClass, CustomizerClass ) );
}
private Class BeanClass = TextReader.class;
private Class CustomizerClass = TextReaderCustomizer.class;
}
```

Although there isn’t a design pattern for it, it’s customary to name a customizer class by postfixing Customizer to the class name. Notice that you named the TextReader customizerTextReaderCustomizer.class. This is a good habit to get into.The programmer has a tremendous amount of freedom when designing customizer classes. There are only two restrictions: The class must inherit from java.awt.Component, so that it can be placed in a Panel or Dialog, and it must implement the java.beans.Customizer interface. The customizer class is given a reference to the target component through a call to the setObject() method. After this point, what the customizer class does is its business, for the most part. Remember, though, that you’ll be required (by the compiler) to acknowledge constrained properties because their accessor methods might throw propertyVetoExceptions. Finally, the java.beans.Customizer interface includes functionality for PropertyChangeListeners. Because the Bean’s container may register itself as a listener with the customizer class, any property updates should be followed by a call to firePropertyChange(). The easiest way to do this is by using ajava.beans.PropertyChangeSupport class as was done when discussing bound properties earlier.

Listing 11.9 shows most of the code for the TextReaderCustomizer class. Some of the AWT specific code was removed for clarity. The full listing is available on the CD-ROM. Take a look at the handleEvent() method. This method is called by AWT when the user enters data. Notice how you were forced to catch ProperyVetoExceptions for the setWidth() accessor? You can also see how the PropertyChangeListener methods are used appropriately. Figure 11.4 shows what the customizer looks like when called up from within Sun’s BeanBox.

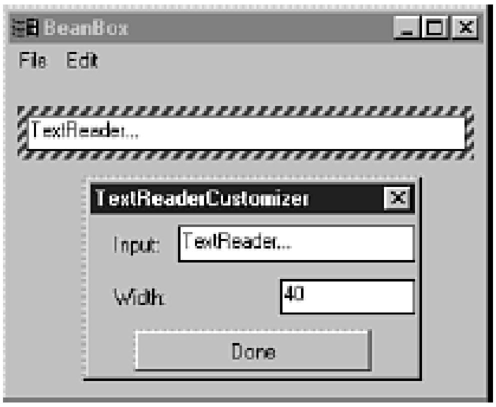
Listing 11.9 TEXTREADERCUSTOMIZER.JAVA—The Code from TextReaderCustomizer.java Showing How to Implement a Customizer Class

```
public class TextReaderCustomizer extends Panel implements Customizer {
```

```
public TextReaderCustomizer() {
setLayout( new BorderLayout() );
}
public void setObject( Object target ) {
component = (TextReader)target;
// generate the User Interface (code removed for clarity)
}
public boolean handleEvent( Event event ) {
if ( event.id == Event.KEY_RELEASE && event.target == InputText ) {
String old_text = component.getInputText();
String text = InputText.getText();
component.setInputText( text );
changeAgent.firePropertyChange( “inputText”, old_text, text );
} else if ( event.id == Event.KEY_RELEASE && event.target == Width ) {
int old_width, width;
old_width = component.getWidth();
try {
width = Integer.parseInt( Width.getText() );
try {
component.setWidth( width );
changeAgent.firePropertyChange( “width”,
ånew Integer( old_width ), new Integer( width ) );
} catch( PropertyVetoException e ) {
// do nothing... wait for acceptable data.
}
} catch( NumberFormatException e ) {
// do nothing... wait for better data.
}
}
return ( super.handleEvent( event ) );
}
public void addPropertyChangeListener( PropertyChangeListener l ) {
changeAgent.addPropertyChangeListener( l );
}
public void removePropertyChangeListener(PropertyChangeListener l) {
changeAgent.removePropertyChangeListener( l );
}
private TextReader component;
private TextField InputText, Width;
private PropertyChangeSupport changeAgent =
ånew PropertyChangeSupport( this );
}
```

FIG. 11.4

Sun’s BeanBox showing the TextReader Bean and its customizer dialog box.



TextReader...

Input: TextReader...

Width: 40

11.8 ENTERPRISE JAVABEANS

A new development on the horizon that utilizes the JavaBeans framework is known as EnterpriseJavaBeans. *Enterprise JavaBeans* is a component model for building and deploying Java in a distributed multitier environment. Enterprise JavaBeans extends the JavaBeans component model to support server components.

Server Components:

Unlike standard JavaBeans, Enterprise JavaBeans are designed to be server components. The advantage of running components on the server is that it enables a multitier construction. In a multitier architecture, much of the logic is placed on the server rather than the client. Creating your application using a multitier design makes it much easier to increase its scalability, performance, and reliability. Using components from the Enterprise Beans allows you to develop extremely flexible multitier apps. These beans can be easily modified as your business rules or economic conditions evolve. In addition, like RMI components, Enterprise JavaBeans can be located anywhere, and the processing is independent of their location.

Adding Component “Run Anywhere”:

Sun has long been touting the “write once, run anywhere” advantages of Java. Using EnterpriseJavaBeans, this concept has been extended. Now not only can Java run on any platform, but Java components can be developed to run on any component execution system. The environment automatically maps the component to the specific system your server is using. So, if you have a server using DCOM, the enterprise system will map to DCOM. If your server is using CORBA, it maps to CORBA, and so on.

Partitioning Your Applications:

Modern object-oriented designs typically split application design into three pieces. These pieces are the user interface, the business logic, and the data. Typically, the server is a relational database management system (DBMS), which the application communicates with but isn’t actually part of the application itself. However, in a multitier architecture, the client application contains only user interface programming. The business logic and data are partitioned and moved into components deployed on one or more servers.

Enterprise JavaBeans:

The result of moving the business and data logic to a server is that your application can take advantage of the power of multithreaded and multiprocessing systems. In addition, the server components can pool and share scarce resources, throughout all the user’s applications. As system demands increase, Enterprise Beans that are heavily used can be replicated and distributed across multiple systems. This means that there is almost no limit to the scalablity of a multitier system. As system resources expire, you can always replicate another system to handle more of the load. In addition to providing enhanced performance, the replication can be used to create many levels of redundancy. This redundancy helps to eliminate any single points of failure.

Reusability and Integration:

Enterprise JavaBeans are accessed through a well-defined interface. The interface allows EnterpriseBeans to be used to create reusable software building blocks, just like regular JavaBeans. A function can be created once and then used repeatedly in whatever application needs that functionality.

Non visual Components:

Enterprise JavaBeans is a server component model for JavaBeans. Enterprise JavaBeans are specialized, non visual JavaBeans that run on a server. Just as with regular JavaBeans, an EnterpriseBean can be assembled with other Beans to create a new application.

Naming:

Enterprise JavaBeans uses another of the new features of Java—JNDI (Java Naming and DirectoryInterface). JNDI defines a mechanism for mapping arbitrary system names to their actual computer location, much like the Internet’s domain name system that allows you to map names like **www.yahoo.com** to the actual computer system the name represents.

11.9 SUMMARY

This chapter covers Important Concepts in Component Models. How JavaBeans specification adopts features common with the other popular component models.

It also covers the basics of designing a JavaBean, Creating and Using Properties,

Using Events to Communicate with Other Components.There are two types of events that components can listen for: multicast events and unicast events.

Customization: Providing Custom PropertyEditors and GUI Interfaces, PropertyEditors and the PropertyEditorManager.It also covers the Enterprise JavaBeans.

11.10 QUESTIONS

1. What are the important Concepts of Component Models in Java Bean.
2. What are the steps to design a Bean.
3. What are different types of properties supported by Java Bean.
4. Write a note on Introspection
5. Write a note on Enterprise Java Bean.

*****

12

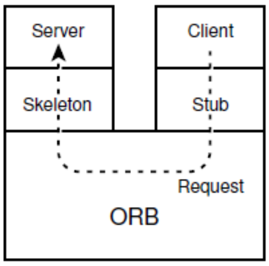
JAVA INTERFACE TO CORBA, JAVA- COM INTEGRATION

Unit Structure

12.1 WHAT IS CORBA?

The Common Object Request Broker Architecture (CORBA) is a tremendous vision of distributed objects interacting without regard to their location or operating environment. CORBA is still in its infancy, with some standards still in the definition stage, but the bulk of the CORBA infrastructure is defined. Many software vendors are still working on some of the features the have been defined.CORBA consists of several layers. The lowest layer is the Object Request Broker, or ORB. The ORB is essentially a remote method invocation facility. The ORB is language-neutral, meaning you can create objects in any language and use the ORB to invoke methods in those objects. You can also use any language to create clients that invoke remote methods through the ORB. There is a catch to the “any language” idea. You need a language mapping defined between the implementation language and CORBA’s

Interface Definition Language (IDL).IDL is a descriptive language—you cannot use it to write working programs. You can only describe remote methods and remote attributes in IDL. This restriction is similar to the restriction in Java that a Java interface contains only method declarations and constants. When you go from IDL to your implementation language, you generate a stub and a skeleton in the implementation language. The stub is the interface between the client and the ORB, while the skeleton is the interface between the ORB and the object (or server). Figure 12.1 shows the relationship between the ORB, an object, and a client wishing to invoke a method on the object.
Fig. 12.1 CORBA clients use the ORB to invoke methods on a CORBA server.



While the ORB is drawn conceptually as a separate part of the architecture, it is often just part of the application. A basic ORB implementation might include the Naming service (discussed shortly) and a set of libraries to facilitate communication between clients and servers. Once client locates a server, it communicates directly with that server, not going through any intermediate program. This permits efficient CORBA implementations. The ORB is both the most visible portion of CORBA and the least exciting. CORBA's big benefit comes in all the services that it defines. Among the services defined in CORBA are

- Lifecycle
- Naming
- Persistence
- Events
- Transactions
- Querying
- Properties

These services are a subset of the full range of services defined by CORBA. The Lifecycle and the Naming services crystallize Sun's visionary phrase "the network is the computer." These services allow you to instantiate new objects without knowing where the objects reside. You might be creating an object in your own program space, or you might be creating an object halfway around the world, and your program will never know it. The Lifecycle service allows you to create, delete, copy, and move objects on a specific system. As an application programmer you would prefer not to know where an object resides. As a systems programmer you need the Lifecycle service to implement this location transparency for the application programmer. One of the hassles you frequently run into in remote procedure call systems is that the server you are calling must already be up and running before you can make the call. The Lifecycle service removes that hassle by allowing you to create an object, if you need to, before invoking a method on it. The Naming service allows you to locate an object on the network by name. You want the total flexibility of being able to move objects around the network without having to change any code. The Naming service gives you that ability by associating an object with a name instead of a network address. The Persistence service allows you to save objects somewhere and retrieve them later. This might be in a file, or it might be on an object database. The CORBA standard doesn't specify which. That is left up to the individual software vendors.

The Event service is a messaging system that allows more complex interaction than a simple message call. You could use the Event service to implement a network-based observerobservablemodel, for example. There are event suppliers that send events, and event consumers that receive events. A server or a client is either push or pull. A push server sends events out when it wants to (it pushes them out), while a push client has a push method and automatically receives events through this method. A pull server doesn't send out events until it is asked—you have to pull them out of the server. A pull client does not receive events until it asks for them. It might help to use the term "poll" in place of "pull." A pull server doesn't deliver events on its own, it gives them out when it is polled. A pull client goes out and polls for events.

The Transaction service is one of the most complex services in the CORBA architecture. It allows you to define operations across multiple objects as a single transaction. This kind of transaction is similar to a database transaction. It handles concurrency, locking, and even rollbacks in case of a failure. A transaction must comply with a core set of requirements that areabbreviated ACID:

- **Atomicity.** A transaction is a single event. Everything in the transaction is either done as a whole or undone. You don't perform a transaction partially.
- **Consistency.** When you perform a transaction, you do not leave the system in an inconsistent state. For example, if you have an airline flight with one seat left, you don't end up assigning that seat to two different people if their transactions occur at the sometime.
- **Isolation.** No other objects see the results of a transaction until that transaction is committed. Even if transactions are

executing simultaneously, they have a sequential order with respect to the data.

- **Durability.** If you commit a transaction, you can be sure that the change has been made and stored somewhere. It doesn’t get lost.

The Transaction service usually relies on an external transaction processing (TP) system. The Object Querying service allows you to locate objects based on something other than name. For instance, you could locate all ships registered in Liberia or all Krispy Kreme donutlocations in Georgia. This service would usually be used when your objects are stored in an object database.

The Properties service allows objects to store information on other objects. A property is like a sticky-note. An object would write some information down on a sticky-note and slap it on another object. This has tremendous potential because it allows information to be associated with an object without the object having to know about it. The beauty of the whole CORBA system is that all of these services are available through the ORB interface, so once your program can talk to the ORB, you have these services available. Of course, your ORB vendor may not implement all of these services yet.

12.2 SUN’S IDL TO JAVA MAPPING METHODS

In order to use Java in a CORBA system, you need a standard way to convert attributes and methods defined in IDL into Java attributes and methods. Sun has proposed a mapping and released a program to generate Java stubs and skeletons from an IDL definition. Defining interfaces in IDL is similar to defining interfaces in Java since you are defining only the signatures (parameters and return values) of the methods and not the implementation of the methods.

IDL Modules:

A module is the IDL equivalent of the Java package. It groups sets of interfaces together in their own namespace. Like Java packages, IDL modules can be nested. The following is an example IDL module definition (shown without any definitions, which will be discussed soon):

```
module MyModule {
// insert your IDL definitions here, you must have at least
// one definition for a valid IDL module
};
```

This module would be generated in Java as a package called MyModule:

```
package MyModule;
```

When you nest modules, the Java packages you generate are also nested. For example, consider the following nested module definition:

```
module foo {
module bar {
module baz {
// insert definitions here
};
};
};
```

Don’t forget to put a semicolon after the closing brace of a module definition. Unlike Java, C, and C++, you are required to put a semicolon after the brace in IDL.

The Java package definition for interfaces within the baz module would bepackage foo.bar.baz;IDL Constants As in Java, you can define constant values in IDL. The format of an IDL constant definition isconst type variable = value;

The type of a constant is limited to boolean, char, short, unsigned short, long, unsigned long, float, double, and string.

Constants are mapped into Java in an unusual way. Each constant is defined as a class with a single static final public variable, called value that holds the value of the constant. This is done because IDL allows you to define constants within a module, but Java requires that constants belong to a class.
Here is an example of an IDL constant definition:

```
module ConstExample {
const long myConstant = 123;
};
```

This IDL definition would produce the following Java definition:

```
package ConstExample;
public final class myConstant {
public static final int value = (int) (123L);
}
```

IDL Data TypesIDL has roughly the same set of primitive data types as Java except for a few exceptions:

- The IDL equivalent of the Java byte data type is the octet.
- IDL supports the String type, but it is called string.

T I P
Sun’s IDL to Java Mapping

- Characters in IDL can have values only between 0 and 255. The JavaIDL system will check your characters to make sure they fall within this range, including characters stored in strings.

- IDL supports 16-, 32-, and 64-bit integers, but the names for the 32- and 64-bit types are slightly different. In IDL, the 32-bit value is called a long, while in Java it is called an int.

  The IDL equivalent of the Java long is the long.
- IDL supports unsigned short, int, and long values. In Java, these values are stored insigned variables. You must be very careful when dealing with large unsigned values,

since they may end up negative when represented in Java.

Enumerated Types

Unlike Java, IDL allows you to create enumerated types that represent integer values. TheJavaIDL system turns the enumerated type into a class with public static final values.

Here is an example of an IDL enumerated type:

```
module EnumModule {
enum Medals { gold, silver, bronze };
};
```

This definition would produce the following Java class:

```
package EnumModule;
public class Medals {
public static final int gold = 0,
silver = 1,
bronze = 2;
public static final int narrow(int i)throws
sunw.corba.EnumerationRangeException {
if (gold <= i && i <= bronze) {
return i;
}
throw new sunw.corba.EnumerationRangeException();
}
}
```

Since you are also allowed to declare variables of an enumerated type, JavaIDL creates a holder class that is used in place of the data type. The holder class contains a single instance variable called value that holds the enumerated value. The holder for the Medals enumeration would look like:

```
package EnumModule;
public class MedalsHolder
{
// instance variable
public int value;
// constructors
public MedalsHolder() {
```

```
this(0);
}
public MedalsHolder(int __arg) {
value = EnumModule.Medals.narrow(__arg);
}
}
```

You can create a Medals Holder by passing an enumerated value to the constructor:

```
MedalsHolder medal = new MedalsHolder(Medals.silver);
```

The narrow method performs range checking on values and throws an exception if the argument is outside the bounds of the enumeration. It returns the value passed to it, so you can use it to perform passive bounds checking. For example,

```
int x = Medals.narrow(y);
```

will assign y to x only if y is in the range of enumerated values for Medals; otherwise, it will throw an exception.

Structures:

An IDL struct is like a Java class without methods. In fact, JavaIDL converts an IDL struct into a Java class whose only methods are a null constructor and a constructor that takes all the structure’s attributes. Here is an example IDL struct definition:

```
module StructModule {
struct Person {
string name;
long age;
};
};
```

This definition would produce the following Java class declaration (with some JavaIDL-specific methods omitted):

```
package StructModule;
public final class Person {
// instance variables
public String name;
public int age;
// constructors
public Person() { }
public Person(String __name, int __age) {
name = __name;
age = __age;
}
}
```

Like the enumerated type, a struct also produces a holder class that represents the structure. The holder class contains a single instance variable called value. Here is the holder for the Person structure:

```
package StructModule;
public final class PersonHolder
{
// instance variable
public StructModule.Person value;
Sun’s IDL to Java Mapping
// constructors
public PersonHolder() {
this(null);
}
public PersonHolder(StructModule.Person __arg) {
value = __arg;
}
}
```

Unions

The union is another C construct that didn’t survive the transition to Java. The IDL union actually works more like the variant record in Pascal, since it requires a “discriminator” value. An IDL union is essentially a group of attributes, only one of which can be active at a time. The discriminator indicates which attribute is in use at the current time. A short example should make this a little clearer. Here is an IDL union declaration:

```
module UnionModule {
union MyUnion switch (char) {
case ‘a’: string aValue;
case ‘b’: long bValue;
case ‘c’: boolean cValue;
default: string defValue;
};
};
```

The character value in the switch, known as the discriminator, indicates which of the three variables in the union is active. If the discriminator is ‘a’, a Value variable is active. Since Java doesn’t have unions, a union is turned into a class with accessor methods for the different variables and a variable for the discriminator. The class is fairly complex. Here is a subset of the definition for the MyUnion union:

```
package UnionModule;
public class MyUnion {
// constructor
public MyUnion() {
// only has a null constructor
}
// discriminator accessor
public char discriminator() throws
sunw.corba.UnionDiscriminantException {
// returns the value of the discriminator
}
// branch constructors and get and set accessors
public static MyUnion createaValue(String value) {
// creates a MyUnion with a discriminator of ‘a’
}
public String getaValue() throws
sunw.corba.UnionDiscriminantException {
// returns the value of aValue (only if the discriminator is ‘a’ right
now)
}
public void setaValue(String value) {
// sets the value of aValue and set the discriminator to ‘a’
}
public void setdefValue(char discriminator, String value)
throws sunw.corba.UnionDiscriminantException {
// Sets the value of defValue and sets the discriminator.
Although every
// variable has a method in this form, it is only useful when you
have
// a default value in the union.
}
}
```

The holder structure should be a familiar theme to you by now. JavaIDL generates a holder structure for a union. The holder structure for MyUnion would be called MyUnionHolder andwould contain a single instance variable called value.

Sequences and ArraysIDL sequences and arrays both map very neatly to Java arrays. Sequences in IDL may be either unbounded (no maximum size) or bounded (a specific maximum size). IDL arrays are always of a fixed size. Since Java arrays have a fixed size but the size isn’t known at compile time,the JavaIDL system performs runtime checks on arrays to make sure they fit within the restrictions defined in the IDL module.

Here is a sample IDL definition containing an array, a bounded sequence, and an unbounded

sequence:

```
module ArrayModule {
struct SomeStructure {
long longArray[15];
sequence<boolean> unboundedBools;
sequence<char, 15> boundedChars;
};
};
```

The arrays would be defined in Java as

```
public int[] longArray;
public boolean[] unboundedBools;
public char[] boundedChars;
```

Exceptions:

CORBA has the notion of exceptions. Unlike Java, however, exceptions are not just a type of object, they are separate entities. IDL exceptions cannot inherit from other exceptions. Other than that, they work like Java exceptions and may contain instance variables. Here is an example of an IDL exception definition:

```
module ExceptionModule {
exception YikesError {
string info;
};
};
```

This definition would create the following Java file (with some JavaIDL-specific methods removed):

```
package ExceptionModule;
public class YikesError
Sun’s IDL to Java Mapping
extends sunw.corba.UserException {
// instance variables
public String info;
// constructors
public YikesError() {
super(“IDL:ExceptionModule/YikesError:1.0”);
}
public YikesError(String __info) {
super(“IDL:ExceptionModule/YikesError:1.0”);
info = __info;
}
}
```

Interfaces:

Interfaces are the most important part of IDL. An IDL interface contains a set of method definitions, just like a Java interface. Like Java interfaces, an IDL interface may inherit from other interfaces. Here is a sample IDL interface definition:

```
module Interface Module {
interface MyInterface {
void myMethod(in long param1);
};
};
```

IDL classifies method parameters as being either in, out, or inout. An in parameter is identical to a Java parameter—it is a parameter passed by value. Even though the method may change the value of the variable, the changes are discarded when the method returns. An out variable is an output-only variable. The method is expected to set the value of this variable, which is preserved when the method returns, but no value is passed in for the variable (it is uninitialized).

An inout variable is a combination of the two—you pass in a value to the method; if the method changes the value, the change is preserved when the method returns. The fact that Java parameters are in-only poses a small challenge when mapping IDL to Java. Sun has come up with a reasonable approach, however. For any out or inout parameters, youpass in a holder class for that variable. The CORBA method can then set the value instance variable with the value that is supposed to be returned.

Attributes:

IDL allows you to define variables within an interface. These translate into get and set methods for the attribute. An attribute may be specified as read-only, which prevents the generation of a set method for the attribute. For example, if you defined an IDL attribute as attribute long my Attribute;

your Java interface would then contain the following methods

:

```
int getmyAttribute() throws omg.corba.SystemException;
void setmyAttribute() throws omg.corba.SystemException;
```

Methods:

You define methods in IDL like you declare methods in Java, with only a few variations. One of the most noticeable differences is that CORBA supports the notion of changeable parameters. In other words, you can pass an integer variable x to a CORBA method, and that method can change the value of x. When the method returns, x has the changed value. In a normal Java method, x would retain its original value. In IDL, method parameters must be flagged as being in, out, or inout. An in parameter cannot be changed by the method, which is the way all Java methods work. An out parameter indicates value that the method will set, but it ignores the value passed in. In other words, if parameter x is an out parameter, the CORBA method cannot read the value of x, it can only change it. An inout parameter can be read by the CORBA method and can also be changed by it.

Here is a sample method declaration using an in, an out, and an inout parameter:

```
long performCalculation(in float originalValue,
inout float errorAmount, out float newValue);
```

Since Java doesn't support the notion of parameters being changed, the Java-IDL mapping uses special holder classes for out and inout parameters. The IDL compiler already generates holder classes for structures and unions. For base types like long or float, JavaIDL has builtinholders of the form TypenameHolder. For example, the holder class for the long type is calledLongHolder. Each of these holder classes contains a public instance variable called value that contains the value of the parameter.

The other major difference between IDL and Java method declarations is in the way exceptionsare declared. IDL uses the raises keyword instead of throws. In addition, the lists of exceptions are enclosed by parentheses. Here is a sample method declaration that throws several exceptions:

```
void execute() raises (ExecutionError, ProgramFailure);
```

12.3 CREATING A BASIC CORBA SERVER

The interface between the ORB and the implementation of a server is called a skeleton. A skeleton for an IDL interface receives information from the ORB, invokes the appropriate server method, and sends the results back to the ORB. You normally don't have to write the skeleton itself; you just supply the implementation of the remote methods.

Listing 12.1 shows an IDL definition of a simple banking interface. You will see how to create both a client and a server for this interface in JavaIDL.

Creating a Basic CORBA Server

Listing 12.1 Source Code for Banking.idl

```
module banking {
enum AccountType {
CHECKING,
SAVINGS
};
struct AccountInfo {
string id;
string password;
AccountType which;
};
exception InvalidAccountException {
AccountInfo account;
};
exception InsufficientFundsException {
};
interface Banking {
long getBalance(in AccountInfo account)
raises (InvalidAccountException);
void withdraw(in AccountInfo account, in long amount)
raises (InvalidAccountException,
InsufficientFundsException);
void deposit(in AccountInfo account, in long amount)
raises (InvalidAccountException);
void transfer(in AccountInfo fromAccount,
in AccountInfo toAccount, in long amount)
raises (InvalidAccountException,
InsufficientFundsException);
};
};
```

Compiling the IDL Definitions:

Before you create any Java code for your CORBA program, you must first compile the IDLdefinitions into a set of Java classes. The idlgen program reads an IDL file and creates classes for creating a client and server for the various interfaces defined in the IDL file, as well as any exceptions, structures, unions, and other support classes.

To compile Banking.idl, the idlgen command would beidlgen -fserver -fclient Banking.idl idlgen allows you to use the C preprocessor to include other files, perform conditional compilation, and define symbols. If you do not have a C preprocessor, use the -fno-cpp option like this:

```
idlgen -fserver -fclient -fno-cpp Banking.idl
```

The -fserver and -fclient flags tell idlgen to create classes for creating a server and a client, respectively. You may not always need to create both, however. If you are creating a Java client that will use CORBA to invoke methods on an existing C++ CORBA server, you only need to generate the client portion. If you are creating a CORBA server in Java but the clients will be in another language, you only need to generate the server portion.

Using Classes Defined by IDL structs :

When an IDL struct is turned into a Java class, it does not have custom hashCode and equals methods. This means that two instances of this class containing identical data will not be equal.

If you want to add custom methods to these structs, you will have to create a separate class and define methods to convert from one class to the other.

One way to remedy this is to create a class that contains the same information as the IDL structure but also contains correct hashCode and equals methods, as well as a way to convert to and from the IDL-defined structure.

Listing 12.2 shows an Account class that contains the same information as the AccountInfo structure defined in Banking.idl.

Listing 12.2 Source Code for Account.java

```
package banking;
// This class contains the information that defines a banking account.
public class Account extends Object
{
// Flags to indicate whether the account is savings or checking
public String id; // Account id, or account number
public String password; // password for ATM transactions
public int which; // is this checking or savings
public Account()
{
}
public Account(String id, String password, int which)
{
this.id = id;
this.password = password;
this.which = which;
}
// Allow this object to be created from an AccountInfo instance
public Account(AccountInfo acct)
{
this.id = acct.id;
this.password = acct.password;
this.which = acct.which;
}
// Convert this object to an AccountInfo instance
public AccountInfo toAccountInfo()
{
return new AccountInfo(id, password, which);
}
public String toString()
{
return “Account { “+id+”,”+password+”,”+which+” }”;
}
// Tests equality between accounts.
public boolean equals(Object ob)
{
if (!(ob instanceof Account)) return false;
Account other = (Account) ob;
return id.equals(other.id) &&
password.equals(other.password) &&
(which == other.which);
}
// Returns a hash code for this object
public int hashCode()
{
return id.hashCode()+password.hashCode()+which;
}
}
```

JavaIDL Skeletons:

When you create a CORBA server, the IDL compiler generates a server skeleton. This skeleton receives the incoming requests and figures out which method to invoke. You only need to write the actual methods that the skeleton will call.JavaIDL creates an Operations interface that contains Java versions of the methods defined in an IDL interface. It also creates a Servant interface, which extends the Operations interface. The skeleton class then invokes methods on the Servant interface. In other words, when you create the object that implements the remote methods, it must implement the Servant interface for your IDL definition.

This technique of defining the remote methods in an interface that can be implemented by separate object is known as a TIE interface. In the C++ world, and even on some early Java ORBS, the IDL compiler would generate a skeleton class that implemented the remote methods.

To change the implementation of the methods, you would create a subclass of the skeleton class. The subclass technique is often called a Basic Object Adapter, or BOA. The advantage of the TIE interface under Java is that a single object can implement multiple remote interfaces. You can’t do this with a BOA object, because Java doesn’t support multiple inheritances. For example, your implementation for the Banking interface might be declared aspublic class BankingImpl implements BankingServantListing 12.3 shows the full BankingImpl object that implements the BankingServant interface.

Notice that each remote method must be declared as throwing sunw.corba.Exception.

Listing 12.3 Source Code for BankingImpl.java

```
package banking;
import java.util.*;
// This class implements a remote banking object. It sets up
// a set of dummy accounts and allows you to manipulate them
// through the Banking interface.
//
// Accounts are identified by the combination of the account id,
// the password and the account type. This is a quick and dirty
// way to work, and not the way a bank would normally do it,
since
// the password is not part of the unique identifier of the account.
public class BankingImpl implements BankingServant
public Hashtable accountTable;
// The constructor creates a table of dummy accounts.
public BankingImpl()
{
accountTable = new Hashtable();
accountTable.put(
new Account(“AA1234”, “1017”, AccountType.CHECKING),
new Integer(50000)); // $500.00 balance
accountTable.put(new Account(“AA1234”, “1017”,
AccountType.SAVINGS),
new Integer(148756)); // $1487.56 balance
accountTable.put(new Account(“AB5678”, “4456”,
AccountType.CHECKING),
new Integer(77 12)); // $77.32 balance
```

```
accountTable.put(new Account(“AB5678”, “4456”,
AccountType.SAVINGS),
new Integer(32201)); // $322.01 balance
}
// getBalance returns the amount of money in the account (in
cents).
// If the account is invalid, it throws an InvalidAccountException
public int getBalance(AccountInfo accountInfo)
throws sunw.corba.SystemException, InvalidAccountException
{
// Fetch the account from the table
Integer balance = (Integer) accountTable.get(new
Account(accountInfo));
// If the account wasn’t there, throw an exception
if (balance == null) {
throw new InvalidAccountException(accountInfo);
}
// Return the account’s balance
return balance.intValue();
}
// withdraw subtracts an amount from the account’s balance. If
// the account is invalid, it throws InvalidAccountException.
// If the withdrawal amount exceeds the account balance, it
// throws InsufficientFundsException.
public synchronized void withdraw(AccountInfo accountInfo, int
amount)
throws sunw.corba.SystemException, InvalidAccountException,
InsufficientFundsException
{
Account account = new Account(accountInfo);
// Fetch the account
Integer balance = (Integer) accountTable.get(account);
// If the account wasn’t there, throw an exception
if (balance == null) {
throw new InvalidAccountException(accountInfo);
}
// If we are trying to withdraw more than is in the account,
// throw an exception
if (balance.intValue() < amount) {
throw new InsufficientFundsException();
}
// Put the new balance in the account
accountTable.put(account, new Integer(balance.intValue() -
amount));
```

```
}
// Deposit adds an amount to an account. If the account is
invalid
// it throws an InvalidAccountException
public synchronized void deposit(AccountInfo accountInfo, int
amount)
throws sunw.corba.SystemException, InvalidAccountException
{
Account account = new Account(accountInfo);
// Fetch the account
Integer balance = (Integer) accountTable.get(account);
// If the account wasn’t there, throw an exception
if (balance == null) {
throw new InvalidAccountException(accountInfo);
}
// Update the account with the new balance
accountTable.put(account, new Integer(balance.intValue() +
amount));
}
// Transfer subtracts an amount from fromAccount and adds it to
toAccount.
// If either account is invalid it throws InvalidAccountException.
// If there isn’t enough money in fromAccount it throws
// InsufficientFundsException.
public synchronized void transfer(AccountInfo fromAccountInfo,
AccountInfo toAccountInfo, int amount)
throws sunw.corba.SystemException, InvalidAccountException,
InsufficientFundsException
{
Account fromAccount = new Account(fromAccountInfo);
Account toAccount = new Account(toAccountInfo);
// Fetch the from account
Integer fromBalance = (Integer) accountTable.get(fromAccount);
// If the from account doesn’t exist, throw an exception
if (fromBalance == null) {
throw new InvalidAccountException(fromAccountInfo);
}
// Fetch the to account
Integer toBalance = (Integer) accountTable.get(toAccount);
// If the to account doesn’t exist, throw an exception
if (toBalance == null) {
throw new InvalidAccountException(toAccountInfo);
}
```

```
// Make sure the from account contains enough money, otherwise throw
// an InsufficientFundsException.
if (fromBalance.intValue() < amount) {
throw new InsufficientFundsException();
}
// Subtract the amount from the fromAccount
accountTable.put(fromAccount,
new Integer(fromBalance.intValue() - amount));
// Add the amount to the toAccount
accountTable.put(toAccount,
new Integer(toBalance.intValue() + amount));
}
}
```

Server Initialization:

While JavaIDL is intended to be Sun's recommendation for mapping IDL into Java, it was released with a lightweight ORB called the Door ORB. This ORB provides just enough functionality to get clients and servers talking to each other but not much more. Depending on the ORB, the initialization will vary, as will the activation of the objects. For theDoor ORB distributed with JavaIDL, you initialize the ORB with the following line:

```
sunw.door.Orb.initialize(servicePort);
```

The servicePort parameter you pass to the ORB is the port number it should use when listening for incoming clients. It must be an integer value. Your clients must use this port number when connecting to your server. After you initialize the ORB, you can instantiate your implementation object. For example,

```
BankingImpl impl = new BankingImpl();
```

Next, you create the skeleton, passing it the implementation object:

```
BankingRef server = BankingSkeleton.createRef(impl);
```

Finally, you activate the server by publishing the name of the object:

```
sunw.door.Orb.publish("Bank", server);
```

Listing 12.4 shows the complete JavaIDL startup program for the banking server.

Listing 12.4 Source Code for BankingServer.java

```
package banking;
```

```
public class BankingServer
{
// Define the port that clients will use to connect to this server
public static final int servicePort = 5150;
public static void main(String[] args)
{
// Initialize the orb
sunw.door.Orb.initialize(servicePort);
try {
BankingImpl impl = new BankingImpl();
// Create the server
BankingRef server =
BankingSkeleton.createRef(impl);
// Register the object with the naming service as “Bank”
sunw.door.Orb.publish(“Bank”, server);
} catch (Exception e) {
System.out.println(“Got exception: “+e);
e.printStackTrace();
}
}
}
```

12.4 CREATING CORBA CLIENTS WITH JAVAIDL

Since the IDL compiler creates a skeleton class on the server side that receives remote method invocation requests, you might expect that it creates some sort of skeleton on the client side that sends these requests. It does, but the client side class is referred to as a stub. The stub class implements the Operations interface (the same Operations interface implemented by the Servant class). Whenever you invoke one of the stub’s methods, the stub creates request and sends it to the server. There is an extra layer on top of the stub called a reference. This reference object, which is the name of the IDL interface followed by Ref, is the object you use to make calls to the server. There are two simple steps in creating a CORBA client in JavaIDL:

1. Create a reference to a stub using the createRef method in the particular stub.
2. Use the sunw.corba.Orb.resolve method to create a connection between the stub and aCORBA server.

Creating CORBA Clients with JavaIDL

You would create a reference to a stub for the banking interface with the following line:

```
BankingRef bank = BankingStub.createRef();
```

Next, you must create a connection between the stub and a CORBA server by “resolving” it.

Since JavaIDL is meant to be the standard Java interface for all ORBs, it requires an ORBindependentnaming scheme. Sun decided on an URL-type naming scheme of the format:
idl:orb_name://orb_parameters

The early versions of JavaIDL shipped with an ORB called the Door ORB, which is a very lightweight ORB containing little more than a naming scheme. To access a CORBA object using the Door ORB, you must specify the host name and port number used by the CORBAserver you are connecting to and the name of the object you are accessing. The format of this information ishostname:port/object_nameIf you wanted to access an object named Bank with the Door ORB, running on a server at port5150 on the local host, you would resolve your stub this way:

```
sunw.corba.Orb.resolve(
“idl:sunw.door://localhost:5150/Bank”,bank);
```

Remember that the bank parameter is the BankingRef returned by the BankingStub.createRefmethod. Once the stub is resolved, you can invoke remote methods in the server using the stub. Listing 12.5 shows the full JavaIDL client for the banking interface. As you can see, once you have connected the stub to the server, you can invoke methods on the stub just like it was a local object.

Listing 12.5 Source Code for BankingClient.java

```
import banking.*;
// This program tries out some of the methods in the
BankingImpl
// remote object.
public class BankingClient
{
public static void main(String args[])
{
// Create an Account object for the account we are going to
access.
Account myAccount = new Account(
“AA1234”, “1017”, AccountType.CHECKING);
AccountInfo myAccountInfo = myAccount.toAccountInfo();
try {
// Get a stub for the BankingImpl object
BankingRef bank = BankingStub.createRef();
sunw.corba.Orb.resolve(
“idl:sunw.door://localhost:5150/Bank”,
bank);
// Check the initial balance
System.out.println(“My balance is: “+
bank.getBalance(myAccountInfo));
// Deposit some money
bank.deposit(myAccountInfo, 50000);
// Check the balance again
System.out.println(“Deposited $500.00, balance is: “+
bank.getBalance(myAccountInfo));
// Withdraw some money
bank.withdraw(myAccountInfo, 25000);
// Check the balance again
System.out.println(“Withdrew $250.00, balance is: “+
bank.getBalance(myAccountInfo));
System.out.flush();
System.exit(0);
} catch (Exception e) {
System.out.println(“Got exception: “+e);
e.printStackTrace();
}
}
}
```

12.5 CREATING CALLBACKS IN CORBA

Callbacks are a handy mechanism in distributed computing. You use them whenever your client wants to be notified of some event, but doesn’t want to sit and poll the server to see if the event has occurred yet. In a regular Java program, you’d just create a callback interface and pass the server an object that implements the callback interface. When the event occurred, the server would invoke a method in the callback object. As it turns out, callbacks are just that easy in CORBA. You define a callback interface in yourIDL file and then create a method in the server’s interface that takes the callback interface as a parameter. The following IDL file defines a server interface and a callback interface:

```
module callbackDemo
{
interface callbackInterface {
void doNotify(in string whatHappened);
};
interface serverInterface {
void setCallback(in callbackInterface callMe);
};
```

};
Under JavaIDL, the setCallback method would be defined as
void setCallback(callbackDemo.callbackInterfaceRef callMe)
throws sunw.corba.SystemException;

Once you have the callbackDemo.callbackInterfaceRef object, you can invoke itswhatHappened method at any time. At this point, the client and server are on a peer-to-peerlevel. They are each other's client and server.

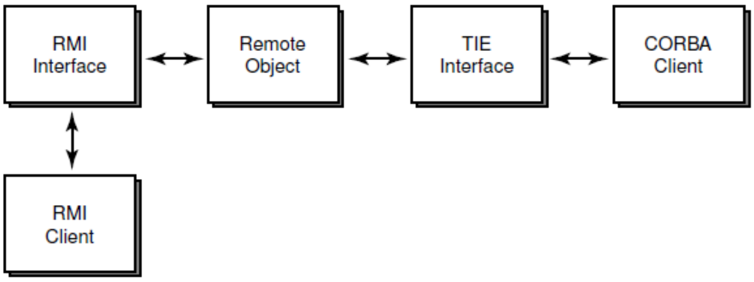
12.6 WRAPPING CORBA AROUND AN EXISTING OBJECT

When you create CORBA implementation objects you are tying that object to a CORBA implementation. While the Servant interface generated by the JavaIDL system goes a long way in separating your implementation from the specifics of the ORB, your implementation method scan throw the CORBA SystemException exception, tying your implementation to CORBA. This is not the ideal situation. You can solve this problem, but it takes a little extra work up front. First, concentrate on implementing the object you want, without using CORBA, RMI, or any other remote interface mechanism. This will be the one copy you use across all your implementations. This object, or set of objects, can define its own types, exceptions, and interfaces.

Next, to make this object available remotely, define an IDL interface that is as close to the object's interface as you can get. There may be cases where they won't match exactly, but you can take care of that. Once you generate the Java classes from the IDL definition, create an implementation that simply invokes methods on the real implementation object. This is essentially the same thing as a TIE interface, with one major exception—the implementation class has no knowledge of CORBA. You can even use this technique to provide multiple ways to access a remote object.

Figure 12.2 shows a diagram of the various ways you might provide access to your implementation object.

Fig.12.2 A single object can be accessed by many types of remote object systems.



While this may sound simple, it has some additional complexities you must address. If your implementation object defines its own exceptions, you must map those exceptions to CORBA exceptions. You must also map between Java objects and CORBA-defined objects. Once again, the banking interface provides a good starting point for illustrating the problems and solutions in separating the application from CORBA. The original banking interface was defined with a hierarchy of exceptions, a genericBankingException, WithInsufficientFundsException and InvalidAccountException as subclasses. This poses a problem in CORBA, since exceptions aren't inherited. You must define a BankingException exception in your IDL file, such as the following:

```
exception BankingException {};
```

In addition, since you probably want the banking application itself to be in the banking package,change the IDL module name to remotebanking.

The implementation for the Banking interface in the remotebanking module must perform two kinds of mapping. First, it must convert instances of the Account object to instances of theAccountInfo object. This may seem like a pain and, frankly, it is. But it's a necessary pain. Ifyou start to intermingle the classes defined by CORBA with the real implementation of the application, you will end up having to carry the CORBA portions along with the application,
even if you don't use CORBA
.

Mapping to and from CORBA-Defined Types:

You can define static methods to handle the conversion from the application data types to the CORBA-defined data types. For example, the banking application defines an Account object. The remotebanking module defines this object as AccountInfo. You can convert between thetwo with the following methods:

```
// Create a banking.Account from an AccountInfo object
public static banking.Account makeAccount(AccountInfo info)
{
return new banking.Account(info.id, info.password,
info.which);
}
// Create an AccountInfo object from a banking.Account object
public static AccountInfo makeAccountInfo(banking.Account account)
{
return new AccountInfo(account.id, account.password,
account.which);
}
```

Your remote implementation of the banking interface needs access to the real implementation,so the constructor for the RemoteBankingImpl object needs a reference to thebanking.BankingImpl object:

```
protected banking.BankingImpl impl;
```

Wrapping CORBA Around an Existing Object

```
public RemoteBankingImpl(banking.BankingImpl impl)
{
this.impl = impl;
}
```

Creating Remote Method Wrappers

Now, all your remote methods have to do is convert any incoming AccountInfo objects to banking. Account objects, catch any exceptions, and throw the proper remote exceptions. Here is the implementation of the remote withdraw method:

```
// call the withdraw function in the real implementation, catching
// any exceptions and throwing the equivalent CORBA exception
public synchronized void withdraw(AccountInfo accountInfo, int amount)
throws sunw.corba.SystemException, InvalidAccountException,
InsufficientFundsException, BankingException
{
try {
// Call the real withdraw method, converting the accountInfo object
// to a banking.Account object first
impl.withdraw( makeAccount(accountInfo), amount);
} catch (banking.InvalidAccountException excep) {
// The banking.InvalidAccountException contains an Account object.
// Convert it to an AccountInfo object when throwing the CORBA exception
throw new InvalidAccountException(
makeAccountInfo(excep.account));
} catch (banking.InsufficientFundsException nsf) {
throw new InsufficientFundsException();
} catch (banking.BankingException e) {
throw new BankingException();
}
}
```

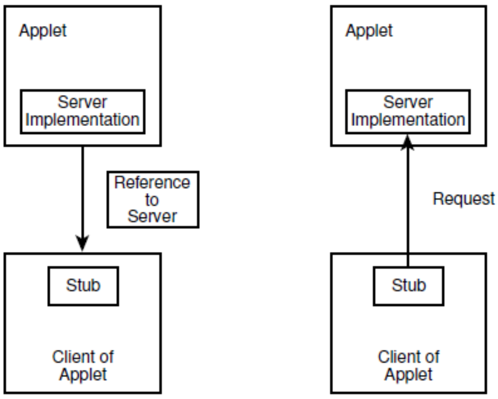
While it would be nice if you could get the IDL-to-Java converter to generate this automatically,it has no way of knowing exactly how the real implementation looks.

12.7 USING CORBA IN APPLETS

Although the full CORBA suite represents a huge amount of code, the requirements for a CORBA client are fairly small. All you really need for a client is the ORB itself. You can access the CORBA services from another location on the network. This allows you to have very lightweight CORBA clients. In other words, you can create applets that are CORBA clients.

The only real restriction on applets using CORBA is that an applet can make network connections back only to the server it was loaded from. This means that all the CORBA services must be available on the Web server (or there must be some kind of proxy set up).Since an applet cannot listen for incoming network connections, an applet cannot be a CORBA server in most cases. You might find an ORB that eludes this restriction by using connections made by the applet. Most Java ORBs available today have the ability to run CORBA servers on an applet for a callback object. For a callback, an applet might create a server object locally and then pass a reference for its server object to a CORBA server running on another machine. That CORBA server could then use the reference to invoke methods in the applet as a client.

Figure 12.3 illustrates how an applet might act as a CORBA server.



Choosing Between CORBA and RMI:

CORBA and RMI each have their advantages and disadvantages. RMI will be a standard part of Java on both the client and server sides, making it a good, cheap tool. Since it is a Java-only system, it integrates cleanly with the rest of Java. RMI is only a nice remote procedure call system; however.CORBA defines a robust, distributed environment, providing almost all the necessary features for distributed applications. Not all of these features have been implemented by most vendors. Most CORBA clients are offered free, but you must pay for the server software. This is the typical pricing model for most Internet software nowadays. If you don’t need all the neat features of CORBA and don’t want to spend a lot of money, RMI might be the right thing for you. Your company might feel that Java is not yet ready for “prime time.” If this is the case but you believe that Java is the environment of the future, you should start working CORBA into your current development plans, if possible.

CORBA is a language-independent system. You can implement your applications in C++ today using many of the Java design concepts. Specifically, keep the application and the user interfaceSeparated and make the software as modular as possible. If you use CORBA between the components of your system, you can migrate to Java by slowly replacing the various components with CORBA-based Java software. If you are a programmer trying to convince your skeptical management about the benefits of Java, use CORBA to make a distributed interface into one of your applications (hopefully you have a CORBA product for the language your application is written in). Next, write a Java applet that implements the user interface for your application using CORBA to talk to the real application. You have instantly ported part of your application to every platform that can run a Java-enabled browser. Hopefully, your applet will perform as well as the old native interface tithe application.

This same technique will open your existing CORBA applications to non-traditional devices like cellular phones and PDAs. If you aren’t ready to support those devices yet, at least you now have a pathway.

12.8 A SIGNIFICANT EXTENSION

Microsoft’s Java Virtual Machine, found in Java-enabled versions of Internet Explorer, contains several Windows-specific extensions that are not a part of standard Java. One of the most significant extensions is the integration with the Component Object Model, or COM.

12.9 A BRIEF OVERVIEW OF COM

Many people have trouble distinguishing between OLE and ActiveX. In fact, many people say that ActiveX is just a fancy name for OLE. The reason for the confusion is that ActiveX andOLE are both based on COM.COM provides a way for objects to communicate, whether they are in the same program, differentprograms on the same machine, or different programs on different machines. The network version of COM, called DCOM (Distributed COM), has only come into production with the introduction of NT 4.0. A version of DCOM for Windows

95 is also available from Microsoft’s Web site at http://www.microsoft.com/oledev. DCOM should be embedded in future versions of Windows.

The goal of COM is not just to allow objects to communicate, but to encourage developers to create reusable software components. Yes, that is one of the stated goals of object-oriented development, but COM goes beyond a typical object-oriented programming language like Java. In Java or C++, you reuse a component by including it in your program. COM allows you to use components that are present on your machine, but are not specifically a part of your program. In fact, a successful COM object is used by many programs. The key here is that there is one copy of the COM object’s code, no matter how many different programs use the object. Although you can arrange your Java classes so that there is no more than one copy of a particular class, you have to do it manually. With COM, this reuse is automatic. Microsoft has embraced the notion of component-based software and has taken great strides to implement applications as a collection of reusable components. Internet Explorer, for instance, is implemented as a collection of components, with just a small startup program. The Java Virtual Machine in Internet Explorer, for instance, is separate components that can be used bother applications. Interfaces are the fundamental foundation of COM. If you understand Java interfaces, you understand COM interfaces. In Java, you can define a set of methods for a class and then also add other definitions by saying that an object implements a certain interface. In other words, in Java, the set of methods implemented by a class is determined by both the class definition and the interfaces it implements. COM doesn’t support the notion of classes, only interfaces. These of methods implemented by a COM object is determined only by the interfaces it implements. Each interface has a unique identifier, called its GUID (Globally Unique Identifier), which is a40-digit number (160 bits) that is generated such that there should never be two identicalGUIDs. One of the important aspects of COM is that after you distribute software that presents a particular interface (with its unique GUID), you should never change that interface. You can’t add to it, you can’t remove things from it; you can’t change the method signatures. If someone else is using one of your components and you change the interface in the next release, you’ll break his software. If you need to change an interface, just create a new one instead.GUIDs come from the DCE RPC standard where they are known as Universally UniqueIDentifiers (UUID).

COM interfaces are defined by the COM Interface Definition Language (IDL), which is a superset of the DCE RPC IDL (an existing standard for remote procedure calls). Don’t confuse COM IDL with CORBA IDL. Although they perform the same function, their syntax is very different. You can also use

OLE's Object Definition Language (ODL) to define COM interfaces.

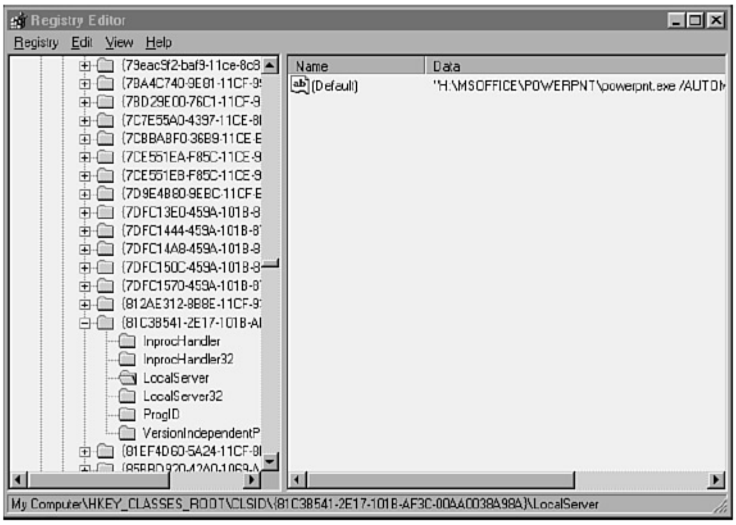
Microsoft offers two different compilers for compiling interfaces. To compile IDL files, useMIDL, this comes with the Win32 SDK, or the Platform SDK. To compile ODL files, useMKTYPLIB, which comes with Visual J++, or the ActiveX SDK.A COM object is accessed one of three basic ways: as an in-proc server, a local server, or a remote server. When you use an in-proc server object, it runs in the same address space as your program. A local server object runs as a separate program on the same machine, while a remote server object runs on a different machine. Not only do COM interfaces have a unique identifier, so do COM objects. The unique identifier for an object is called its Class ID (CLSID). A CLSID is really a GUID, but it serves a specific purpose so it is given a separate name. These CLSID values are stored in the Windows Registry file and are used to find the particular DLL or EXE file that implements an object. Figure 12.1 shows a Registry entry for a CLSID that happens to be for a PowerPoint application. The various sub keys, such as LocalServer and InprocHandler, indicate which DLL or EXE file to use when the PowerPoint object is used as a local server or an in-proc server. Other than the actual functions they perform, the only difference between OLE and ActiveX is that they use different interfaces. All OLE and ActiveX interfaces are defined and implemented using COM.

12.10 DEFINING COM INTERFACES

To create a COM object, you must first create an interface using either IDL or ODL. Because the MKTYPLIB utility (the ODL compiler) comes with Visual J++, the examples in this chapter use ODL instead of IDL.

FIG. 12.4

The Registry entry for the LocalServer of a CLSID indicates the EXE file that



Name	Data
(Default)	"H:\MSOFFICE\POWERPNT\powerpnt.exe /AUTOM

My Computer\HKEY_CLASSES_ROOT\CLSID\{81C3B541-2E17-101B-AF3C-00AA0038A98A}\LocalServer provides a specific COM interface.

Listing 12. 6 shows a sample ODL file. There are a number things in this file that may seem foreign. By taking them one at a time, you see that things are not as complicated as they seem.

Listing 12.6 Source Code for JavaObject.odl

```
// JavaObject.odl
// First define the uuid for this type library
[
uuid(D65E5380-6D58-11d0-8F0B-444553540000)
]
// Declare the type library
library LJavaObject
{
// Include the standard set of OLE types
importlib(“stdole32.tlb”);
// Define the uuid for an odl interface that is a dual interface
// A dual interface is the most flexible because it supports the
// normal interface calling mechanism and also dynamic calling.
//
[ odl, dual, uuid(D65E5381-6D58-11d0-8F0B-444553540000) ]
// Declare the IJavaObject interface (dual interfaces must inherit
// from the IDispatch interface)
interface IJavaObject : IDispatch
{
// Declare the reverseString method that takes a string as input
// and returns a string
HRESULT reverseString( [in] BSTR reverseMe,
[out, retval] BSTR *reversed );
// Declare the square method that takes an integer and returns
// an integer
HRESULT square( [in] int squareMe,
[out, retval] int *squared );
}
// Declare a class that implements the IJavaObject interface
[ uuid(10C24E60-6D5D-11d0-8F0B-444553540000) ]
coclass JavaObject
{
interface IJavaObject;
}
};
```

First of all, when you create a set of interfaces with ODL, you compile them into a type library.

A type library is to an ODL file what a .CLASS file is to the Java source. A type library must have its own GUID, so the following statement declares the library and its GUID (remember that a GUID is another name for UUID):

```
[ uuid(D65E5380-6D58-11d0-8F0B-444553540000)]
library LJavaObject
```

It may seem a little awkward, but you define an object’s GUID just ahead of the objectitself. In other languages, you usually start off with the object itself. The importlib statement is similar to the import keyword in Java. In the previous example, itis importing a set of standard OLE definitions. After the importlib comes the definition of theIJavaObject interface:

```
[ odl, dual, uuid(D65E5381-6D58-11d0-8F0B-444553540000) ]
interface IJavaObject : Idispatch
{
```

Notice that the uuid keyword is accompanied by the odl and dual keywords. The bracketed area where you normally define the uuid is used for any kind of attribute. You almost always find uuid there, because every interface and class must have its own unique identifier. Whenever you define an interface in ODL, you use the odl keyword. The dual keyword specifies that the interface is a dual interface.COM has two different ways of invoking methods—through a lookup table or through a dispatch interface. The lookup table is better known as a vtable—a virtual method lookup table, similar to the vtable in C++. The dispatch interface allows you to perform dynamic method invocation. When you use a dispatch interface, there is an extra level of lookup that takes place before the method is invoked. This tends to be slower than a vtable method invocation, but is useful to interpreted languages like Visual Basic. To allow the maximum flexibility, you can implement your classes with both vtable and dispatch interfaces by declaring them as dual interfaces.

The method definitions also look rather strange:

```
HRESULT reverseString( [in] BSTR reverseMe,
[out, retval] BSTR *reversed );
```

Believe it or not, the reverseString method really returns a string, and not the HRESULT valueyou see declared. The HRESULT return value is necessary when creating this dual interface. The actual return value is specified by the [out, retval] attribute for one of the parameters. A parameter with an attribute of [in] is an input parameter, while those with an [out] attribute are output parameters.

The BSTR data type is a basic string and is the common way to represent strings in COM.

There are other ways, but BSTR is compatible with OLE and also Visual Basic. You may be pleasantly surprised to know that the Java-COM compiler translates the definition ofreverseString into this rather simple method declaration:

```
public String reverseString(String reverseMe)
throws com.ms.com.ComException
```

The definition of the JavaObject class at the bottom of the ODL file tells what interfaces a JavaObject class implements. In this case, there is a single interface: IJavaObject. If you look at different classes available on your system, especially OLE servers, you will find that most classes implement several different interfaces.

12.11 COMPILING AN ODL FILE

The MKTYPLIB program that comes with Visual J++, and also the ActiveX SDK, compiles anODL file into a type library file, which has an extension of .TLB. The MKTYPLIB command to compile JavaObject.odl is:

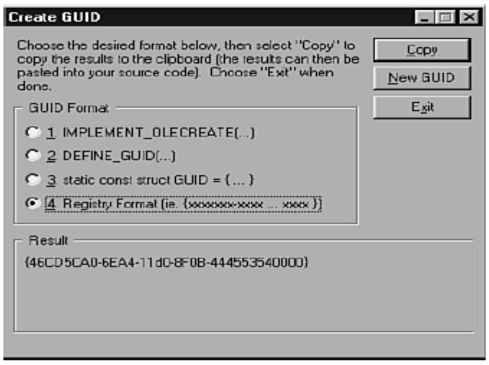
```
mktyplib JavaObject.odl
```

By default, MKTYPLIB uses the C preprocessor, which allows you to use #include and other preprocessor directives. Unfortunately, this only works if you have a C preprocessor. If you don’t, you must use the /nocpp option, like this:

```
mktyplib /nocpp JavaObject.odl
```

12.12 GENERATING A GUID

The JavaObject ODL file contains three different GUIDs. You don’t have to make these values up (in fact, you shouldn’t). Instead, Visual C++ and the ActiveX SDK (and probably other packages, too) come with a tool called GUIDGEN which randomly generates these values. It can Creating COM Objects in Java format them in a number of ways and can even copy them to the Clipboard automatically so you can paste them into your source code. Figure 12...2 shows a sample GUIDGEN session.



Choose the desired format below, then select "Copy" to copy the results to the clipboard (the results can then be pasted into your source code). Choose "Exit" when done.

○ 1 IMPLEMENT_OLECREATE(...)
○ 2 DEFINE_GUID(...)
○ 3 static const struct GUID = { ... }
● 4 Registry Format (ie. {xxxxxxxx-xxxx ... xxxx })

{46CD5CA0-6EA4-11d0-8F0B-444553540000}

12.13 CREATING COM OBJECTS IN JAVA

To create a Java object that implements one or more COM interfaces, you need to create special wrapper classes using the JAVATLB command. For example, to create the Wrapper classes for the information in JavaObject.tlb (which was compiled from JavaObject.odl), theJAVATLB command would be:

```
javatlb JavaObject.tlb
```

For the JavaObject.tlb file, JAVATLB creates an interface called IJavaObject and a Java class called JavaObject. These classes belong to the package javaobject (all lowercase) and are placed in the \WINDOWS\JAVA\TRUSTLIB directory. Remember that packages require their own subdirectory, so if you look in \WINDOWS\JAVA\TRUSTLIB, you will find a directory calledjava object that contains IJavaObject.class and JavaObject.class.

After the wrappers have been created, you only need to fill in the appropriate methods. Listing 12.7 shows the JavaObjectImpl class that implements the methods in the IJavaObject interface.

Listing 12...7 Source Code for JavaObjectImpl.java

```
import com.ms.com.*;
import javaobject.*;
public class JavaObjectImpl implements IJavaObject
{
```

FIG. 12..2
The GUIDGEN tool
automatically generates
GUID values for you.

continues

```
public String reverseString(String in)
throws ComException
{
StringBuffer buff = new StringBuffer();
// Start at the end of the input string and add characters
// to the string buffer. This puts the reverse of the string
// into the buffer.
for (int i=in.length()-1; i >= 0; i—) {
buff.append(in.charAt(i));
}
// Return the contents of the buffer as a new string
return buff.toString();
}
public int square(int val)
throws ComException
{
// Return the square of val
return val * val;
}
}
```

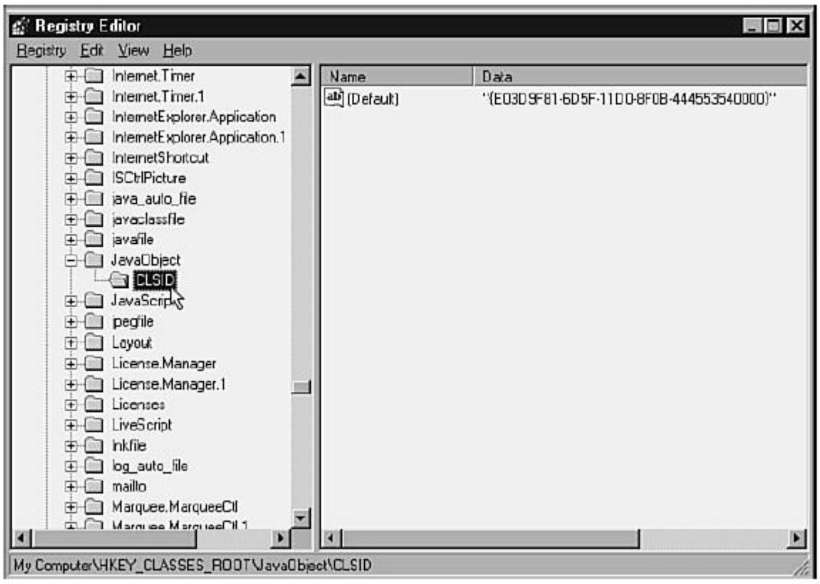
After you have compiled JavaObjectImpl (which you must compile with the Microsoft Java compiler, JVC), use the JAVAREG tool to put information about JavaObjectImpl into the system

Registry. COM uses the Registry to locate COM objects and to find out how to run the server for a particular object. The following command registers JavaObjectImpl and gives it a ProgIDof JavaObject:

```
JAVAREG /register /class:JavaObjectImpl /progid:JavaObject
```

The ProgID value is a simple name that other programs like Visual Basic can use to locate the JavaObject class. JAVAREG creates an entry in the HKEY_CLASSES_ROOT section of the Registrycalled JavaObject, which contains a subkey called CLSID containing the class ID (GUID) forJavaObjectImpl. Figure 12..3 shows this Registry entry, as shown by the REGEDIT command.JAVAREG also creates an entry under CLSID in HKEY_CLASSES_ROOT. The entry's key is theCLSID for JavaObjectImpl (the same CLSID contained in the ProgID entry for JavaObject).

Figure 12..4 shows the entries made under the CLSID as shown by REGEDIT.

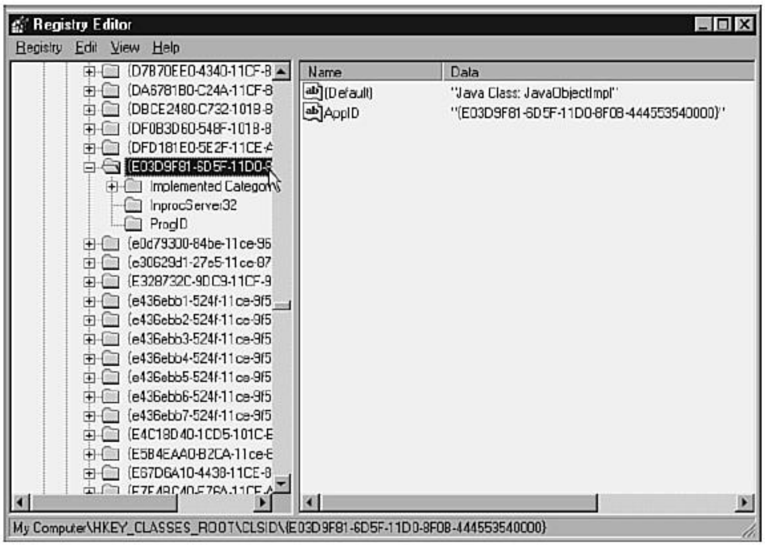


Name	Data
(Default)	"{E03D9F81-6D5F-11D0-8F0B-444553540000}"

My Computer\HKEY_CLASSES_ROOT\JavaObject\CLSID

The final step in making your class available to the rest of the world is to copy theJavaObjectImpl.class file into \WINDOWS\JAVA\TRUSTLIB.

Fig. JAVAREG makes a number of entries under the CLSID key.



Name	Data
(Default)	"Java Class: JavaObjectImpl"
AppID	"{E03D9F81-6D5F-11D0-8F08-444553540000}"

My Computer\HKEY_CLASSES_ROOT\CLSID\{E03D9F81-6D5F-11D0-8F08-444553540000}

12.14 CALLING JAVA COM OBJECTS FROM VISUAL BASIC

If you have run JAVAREG to register your class, and you have copied the class to theTRUSTLIBdirectory, you should now be able to access your class from other programs. You can create a simple Visual Basic application to access this class. In the declaration section for the VB application,insert the following statement:

Dim javaob as ObjectNext, the Form_Load subroutine, which is called when the VB application starts up, should look like this:

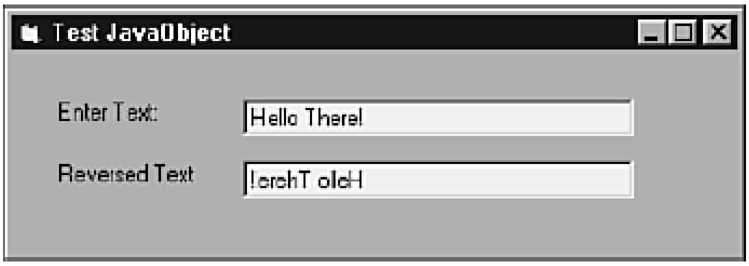
```
Private Sub Form_Load()
Set javaob = CreateObject(“JavaObject”)
End Sub
```

The JavaObject string is the ProgID for the object. If you used something else as the ProgIDwhen you ran JAVAREG, you would use that name here.Now you can make use of the methods in the JavaObject class. In this example VB application,there are two text fields: Text1 and Text2. The following subroutine takes the text from Text1,runs it through the reverseString method in JavaObject, and puts the resulting text in Text2:

```
Private Sub Text1_Change()
```

```
Text2.Text = javaob.reverseString(Text1.Text)
End Sub
```

Figure 12..5 shows this Visual Basic application in action.

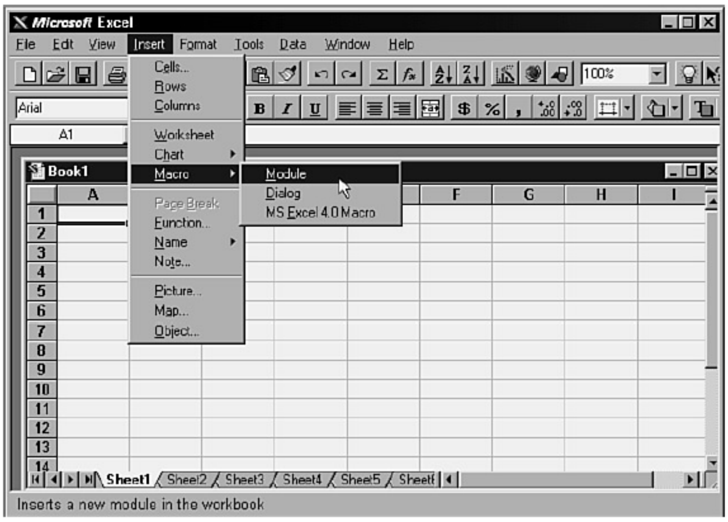


Enter Text: Hello There!

Reversed Text: !erehT olleH

12.15 CALLING JAVA OBJECTS FROM EXCEL

Microsoft Excel and other Microsoft Office products have their own version of Visual Basicbuilt-in. This means, of course, that you can also access Java objects from Excel!To create an Excel function, start Excel and choose Insert, Macro, Module from the main menu, as shown in Figure 12..6.



Inserts a new module in the workbook

The function must access the JavaObject class and then call reverseString. Listing 12..3shows the Reverser$ function.

Listing 12..3 Reverser$ Function from ExcelDemo.xls

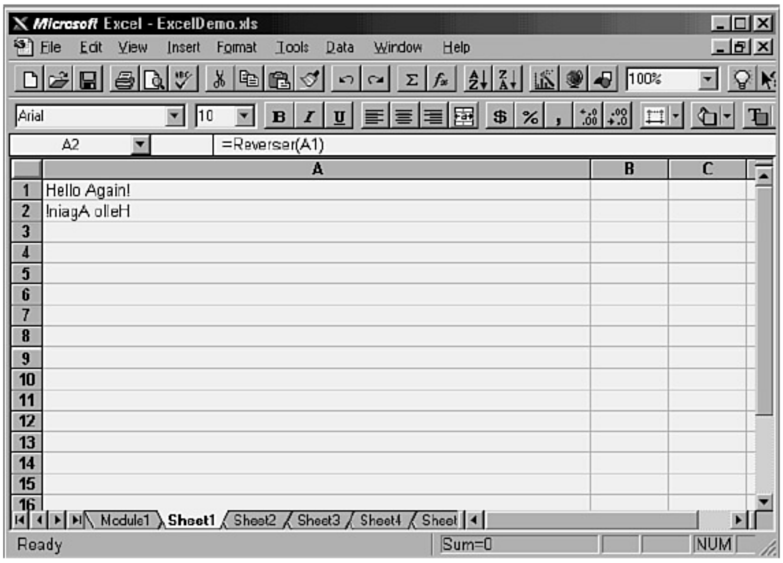
```
Function Reverser$(reverseMe$)
Dim javaob As Object
Set javaob = CreateObject("JavaObject")
Reverser$ = javaob.reverseString(reverseMe$)
End Function
```

After you have defined this function, you can use it in your spreadsheet. For example, assume that you want to take the information in cell A1 in the spreadsheet, reverse it, and place the results in cell A2. Just go to cell A2 and type the following formula:

=Reverser(A1)

Notice that there is no $ at the end of Reverser in this case. Now, any text you type in A1 will automatically appear reversed in A2. Figure 12.7 shows a sample spreadsheet.

FIG. 12.6To create an Excel function, you need to insert a code module.


Microsoft Excel - ExcelDemo.xls
File Edit View Insert Format Tools Data Window Help
100%
Arial
10
A2
Module1 Sheet1 Sheet2 Sheet3 Sheet4 Sheet
Ready
Sum=0
NUM

A2 =Reverser(A1)

	A	B	C
1	Hello Again!		
2	IniagA olleH		

12.16 CALLING COM OBJECTS FROM JAVA

Just as you can access Java objects via COM, Java objects can access other COM-aware objects.

This really opens up possibilities for you on the Windows platform. If you want to create a graph, you can use Excel's graphing capabilities. If you want to create a neatly formatted print out, you can create a document in Word and print it. The best part is you don't have to go through the pain of creating an ODL file, as long as you can get the type library for the application you want to use.

In the case of Microsoft Word, version 7, you can get the type library for free from Microsoft at **http://www.microsoft.com/WordDev/Articles/wb70endl.htm.**

After you have a type library, run JAVATLB on the library to create the Java wrappers. For example,the JAVATLB command for the Word 7 type library would be:javatlb wb70en32.tlb

This command creates a WordBasic interface class that you can use. Because Word runs as a local server object, you have to do a little more work to access it.The LicenseMgr object can access a local server object given the CLSID of the object. You don’t want to look up the Word.Basic CLSID yourself and put it in the source code. Instead, you can access the system Registry from your program and discover the CLSID at runtime. The RegKey class gives you access to the registry. Given a RegKey object, you retrieve subkeysby calling the RegKey constructor with the parent key. For instance, you want the keyCLASSES_ROOT\Word.Basic\CLSID. You first need the RegKey for “CLASSES_ROOT.” Fromthat, you create a RegKey for Word.Basic, which is then used to create the RegKey for CLSID.

After you have the RegKey you want, you access the default value by calling the enumValuemethod. Listing 12..8 shows a demo program that runs Word 7.0, creates a simple “HelloWorld” document, and prints it.

Listing 12.8 Source Code for WordDemo.java

```
import wb70en32.*;
import com.ms.com.*;
import com.ms.lang.*;
public class WordDemo extends Object
{
public static void main(String[] args)
{
try {
// Get the Registry Key for CLASSES_ROOT
RegKey root = RegKey.getRootKey(RegKey.CLASSES_ROOT);
// From CLASSES_ROOT, get the key for Word.Basic
RegKey wbkey = new RegKey(root,
“Word.Basic”, RegKey.KEYOPEN_READ);
// From Word.Basic, get the CLSID
RegKey clsid = new RegKey(wbkey, “CLSID”,
RegKey.KEYOPEN_READ);
// Retrieve the CLSID from the CLSID key (it’s the default value)
String classID = ((RegKeyEnumValueString)clsid.
enumValue(0)).value;
// Create a License Manager for accessing local server objects
ILicenseMgr lm = (ILicenseMgr) new LicenseMgr();
// Get a reference to WordBasic
WordBasic wb = (WordBasic)
lm.createInstance(classID,
null, ComContext.LOCAL_SERVER);
// Create a new file
wb.FileNewDefault();
// Insert some text
```

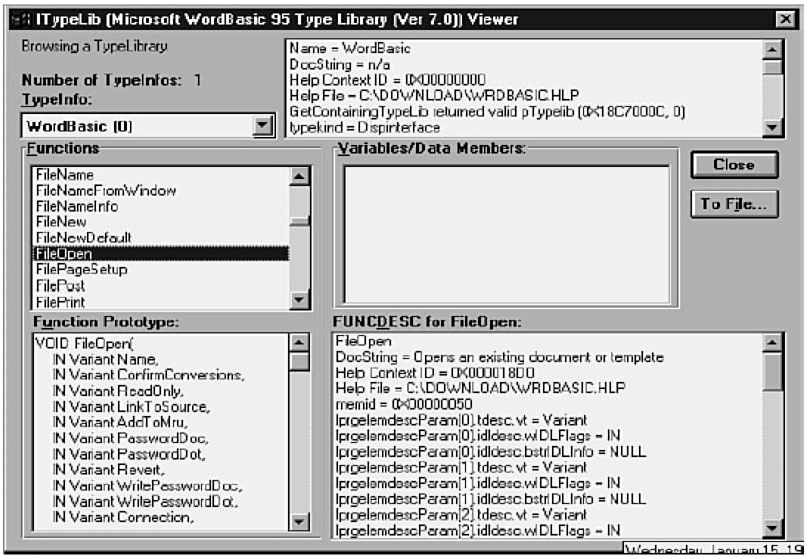
```
wb.Insert("Hello World!");
wb.InsertPara();
wb.Insert("Hi there!");
// Print the text
wb.FilePrintDefault();
```

continues

```
} catch (Error e) {
e.printStackTrace();
}
}
}
```

Remember to use the jview command to run programs in the MS Java environment, rather than java.If you want to see the methods available from the WordBasic object, use the OLE object viewer that comes with Visual J++ (OLE2VIEW) or the ActiveX SDK (OLEVIEW). Figure 12..8 shows the OLE2VIEW display of one of the methods in the WordBasic object. One of the things you are bound to encounter with the WordBasic object, and others, is that some methods have parameters of the variant type. The Java-COM package comes with a Variant object that allows you to pass variant parameters. For example, if you want to call a method that takes two variant parameters, you create two instances of a Variant object. Variant parameters are used when parameters are optional. For this example, assume that the second parameter is optional and the first one is an integer. The sequence of events would go like this:

```
Variant p1 = new Variant(); // Create parameter 1
p1.putInt(5); // Make the parameter value = 5
Variant p2 = new Variant(); // Create parameter 2
p2.noParam(); // Don't pass a value for this parameter
someObject.funMethod(v1, v2); // Call the method
```



```
Name = WordBasic
DocString = n/a
Help Context ID = 0X00000000
Help File = C:\DOWNLOAD\WRDBASIC.HLP
GetContainingTypeLib returned valid pTypelib (0X18C7000C, 0)
typekind = Dispinterface
```

```
VOID FileOpen(
    IN Variant Name,
    IN Variant ConfirmConversions,
    IN Variant ReadOnly,
    IN Variant LinkToSource,
    IN Variant AddToMru,
    IN Variant PasswordDoc,
    IN Variant PasswordDot,
    IN Variant Revert,
    IN Variant WritePasswordDoc,
    IN Variant WritePasswordDot,
    IN Variant Connection,
```

```
FileOpen
DocString = Opens an existing document or template
Help Context ID = 0X000018D0
Help File = C:\DOWNLOAD\WRDBASIC.HLP
memid = 0X00000050
lprgelemdescParam[0].tdesc.vt = Variant
lprgelemdescParam[0].idldesc.wIDLFlags = IN
lprgelemdescParam[0].idldesc.bstrIDLInfo = NULL
lprgelemdescParam[1].tdesc.vt = Variant
lprgelemdescParam[1].idldesc.wIDLFlags = IN
lprgelemdescParam[1].idldesc.bstrIDLInfo = NULL
lprgelemdescParam[2].tdesc.vt = Variant
lprgelemdescParam[2].idldesc.wIDLFlags = IN
```

12.17 SUMMARY

This chapter covers What Is CORBA? Sun’s IDL to Java Mapping Methods .

This chapter also covers how a Basic CORBA Server is created and how CORBA Clients is Created with JavaIDL How the Callbacks are Creating in CORBA .

How CORBA is wrapped around an existing object. How CORBA is used in Applets.

It also covers a brief overview of COM, defines COM Interfaces. How to compile an ODL File, Generating a GUID, Creating COM Objects in Java ,Calling Java COM Objects from Visual Basic, Calling Java Objects from Excel, Calling COM Objects from Java .

12.18 QUESTION

1. What Is CORBA?
2. How will you Create a Basic CORBA Server
3. How will you Create a CORBA Clients with JavaIDL
4. How will you create Callbacks in CORBA
5. Define COM Interfaces
6. How will you create COM Objects in Java
7. How will you call Java COM Objects from Visual Basic
8. How will you call Java Objects from Excel
9. How will you call COM Objects from Java.

*****

13

JAVA MEDIA FRAMEWORK

Unit Structure

13.1 WHAT IS THE JAVA MEDIA FRAMEWORK?

In the first two incantations of Java, programmers were hard-pressed to use Java when they wanted to display complex media types like sound and video. Sure, Java had support for one audio type (au), but playing complex types like MIDI and WAV was not possible. In addition, video and animation could be done by creating slide like presentations; but displaying AVI or MOV files again wasn't possible without a lot of work, and even then it wasn't practical. With the Java Media Framework, Java has come of multimedia age. The Java Media Framework provides the means to present all kinds of interesting media types. The Java Media Framework is an API for integrating advanced media formats into Java, such as video and sound. The mediaframework has several components, including media players, media capture, and conferencing. A key to any of these topics is in providing a timing mechanism, which determines when the next part of the media should be played. In other words, it's important to have a mechanism to keep a video stream playing at the same speed as an accompanying sound stream. Otherwise, you will end up with lips moving and somebody else's words coming out.

13.2 CREATING A MEDIA PLAYER

To understand the Java Media Framework, it is best to jump straight in, by creating an applet that uses a media player. Putting a media player into an applet involves a few basic steps:

1. Create the URL for the media file.
2. Create the player for the media.
3. Tell the player to prefetch.
4. Add the player to the applet.
5. Start the player.

The first step is something you've done a dozen times already in this book. The next steps are what you need to do to create the player itself. To do this, you utilize the Manager class. The Manager class is actually the hub for getting both the timebase and the players. You need to give the manager the URL where the media file is located.

The first task is to create an URL for the media file and then to create the player. In the BasicPlayer class, this happens in the init() method, as shown in Listing 13.1.

Listing 13.1 Creating a Player and Its Associated URL

```
try {
mediaURL = new URL(getDocumentBase(), mediaFile);
player = Manager.createPlayer(mediaURL);
}
catch (IOException e) {
System.err.println("URL for "+mediaFile+" is invalid");
}
catch(NoPlayerException e) {
System.err.println("Could not find a player for " + mediaURL);
}
}
```

By now you should be familiar with how the URL code is created in the second line of Listing 13.1. But just for review, the getDocumentBase() method returns the URL where the applet was originally loaded from, and with mediaFile representing the relative URL where the media file is located, the mediaURL ends up pointing to the fully qualified URL of the media file.The next line is really the one you're interested in. The manager is asked to return the player that knows how to deal with the URL you've provided.

Note that there are two kinds of exceptions you need to catch in this situation. First, the URL constructor will throw an IOException if the URL isn't valid. Because we are using the documentBase of the applet, it's highly unlikely that this exception will ever occur for this particular example, but you need to catch this exception.

The second kind of exception is the NoPlayerException. This is a new one from the media package. This particular exception is thrown when the manager knows of no player that is designed for the media type the URL points to. For instance, if you had pointed to a .wav file and there was no player for the .wav file, the NoPlayerException would get thrown. Prefetching the Media.

The next step in the process is to prefetch the media. Prefetching causes two things to happen. First, the player goes through a process called realization. It then starts to download the media file so that some of it can be cached. This reduces the latency time before the player can start actually playing the media. Later in this chapter, we will go more into the states the player can go through, such as prefetch and realize. For now, it's sufficient to recognize that you need to ask the player to prefetch. As Listing 13.2 shows, in your most basic player you will do this in the start() method.

Listing 13.2 in the start Method, We Prefetch the Media We Are Going
to Play

```
public void start() {
if (player != null)
//prefetch starts the player.
//Prefetch returns immediately, just like getImage
player.prefetch();
}
```

As you can see, all you need to do to prefetch the media is call the prefetch method on the
player.

Adding the Player to Your Application:

Adding the player to the application is actually kind of tricky. The player itself is not an AWTcomponent. So you don't add the player itself, but its visual representation. To get the visual component, player has a method called getVisualComponent(). However, there is a catch—before you can get the visual component, the player must first be realized. Again, we will talk more about this later, but for now, it's important to understand that just as an image isn't valid right after you call get Image(), the player is not valid until it has been realized.How then do you know whether the component has been realized? Well, player has a method called getState() that returns the state of the current player. If you wanted to, you could constantly check the state, and when the state of the player was finally realized, you could ask for the visual component.

Fortunately, there is a more efficient way. As you might recall, Image has an interface calledImageObserver. The media API has a similar method, called ControllerListener.ControllerListener has one method—controllerUpdate(ControllerEvent). We can use thecontrollerUpdate method to know when the media has been fetched, as shown in Listing 13.3.

Listing 13.3 The controllerUpdate Method Is Called Each Time theController Changes State

```
public synchronized void controllerUpdate(ControllerEvent event)
{
if (event instanceof RealizeCompleteEvent)
{
// Once the player has been realized add the
//visual component to the screen
if ((visualComponent = player.getVisualComponent()) != null)
add(“Center”, visualComponent);
// draw the component
validate();
}
}
```

The controllerUpdate() method is called each time the state of the controller changes. As we will see later in this chapter, the player is itself a controller and it can go through many different state changes. To differentiate between these changes, there are many events that can be generated. In this case, what we are looking for is the RealizeCompleteEvent. To determine which event has been received, we use the instanceof operator to evaluate whether the current event is of the class RealizeCompleteEvent. If it is, we know that the player has been realized, and we can now request the visual component from the player and add it to the application. It is possible that the getVisualComponent() method will return a null. This happens when there is no visual representation of the media. For instance, a player for an audio file might not need to have a visual component. Obviously, that means you need to test for the null condition before adding it to the applet.

In addition, to make sure that the component is actually represented in the applet, you do need to make sure that the applet is validated. The validate() method forces any container, including an apple, to make sure that all the components that have been added are actually present on the screen.

Registering the Applet as a Listener:

Before we can actually utilize the new controllerUpdate() method, we need to back up a step. You’re probably already wondering how the player knows to inform the application about the changes. The answer is that it doesn’t know right away. To have the player call controllerUpdate(), you must first register your application with the player. In traditional event listener fashion, this is done using player’s addControllerListener() method. Just like all java.awt.event listeners, after a component has been registered as a listener, its listener method will be called (in this case, controllerUpdate()) any time an event occurs. For the current purposes, you will add the addControllerListener code to the init() method of the applet, so the whole init() method now looks as shown in Listing 13.4.

Listing 13.4 In the init() Method, We Add this as a Listener to the Player

```
public void init() {
String mediaFile = null;
URL mediaURL = null;
setLayout(new BorderLayout());
if ((mediaFile = getParameter(“file”)) == null) {
System.err.print (“Media File not present.”);
System.err.println(“ Required parameter is ‘file’”);
}
else {
try {
mediaURL = new URL(getDocumentBase(), mediaFile);
player = Manager.createPlayer(mediaURL);
if(player != null) {
//tell the player to add this applet as a listener
player.addControllerListener(this);
}
else
System.err.println(“failed to create player for “ + mediaURL);
}
catch (IOException e) {
System.err.println(“URL for “+mediaFile+” is invalid”);
}
catch(NoPlayerException e) {
System.err.println(“Could not find a player for “ + mediaURL);
}
}
}
```

Starting the Player :

The next step in the process is to tell the player to start. However, just as with requesting the visual component, the player cannot be started until it has reached a certain state—namely, the player must have managed to complete the prefetching that we talked about earlier. Technically speaking, as soon as the player has been realized, you can start it; however, it's unwise to do so because it's likely that the media will play choppily. The prefetching of the media allows the player to streamline some of the media so that it can smoothly play the whole thing.

To determine when the prefetch has been completed, we can look for a PrefetchCompletedEvent. After this event has been generated, the prefetching is complete. To utilize this knowledge, we will add the code shown in Listing 13.5 to the controllerUpdatemethod.

Listing 13.5 Start the Player After the Prefetch Is Complete

```
if (event instanceof PrefetchCompleteEvent) {
// start the player once it's been prefetched
player.start();
}
```

Cleaning Up and Stopping the Player :

Being good public programmers, we must use the stop() method of the applet to clean up after ourselves. As you will recall, the stop() method is called when the browser leaves the current Web page. After the browser leaves the page, we should stop playing the media. Stopping the player is easy to do; we just use the stop() method on the player. However, there is one additional step we should take—removing the media from memory. As you can well imagine, keeping a 2 or 3MB video that is no longer needed in memory is very wasteful and will eventually bring the whole system to its knees. Fortunately, player has a method for just this purpose: deallocate(). As soon as you know that you no longer need a media, you should tell the player to deallocate it so that it can be garbage-collected. Using both the player's stop() and the deallocate() methods, you can create the applet's stop() method. Listing 13.6 shows what the complete stop() method should look like.

Listing 13.6 Stop the Player and Deallocate the Media in the stop() Method

```
public void stop()
{
if (player != null)
{
// Stop media playback
player.stop();
//release resources for the media
player.deallocate();
}
}
```

Putting It All Together :

Now, you can put everything you have just learned together into a complete player. Listing 13.7shows what the complete player looks like.

Listing 13.7 BasicPlayer.java—A Complete Media Player Using the Java Media Framework Classes

```
import java.applet.*;
import java.awt.*;
import java.net.*;
import java.io.*;
import javax.media.*;
/**
* A basic media player Applet
*/
public class BasicPlayer extends Applet implements ControllerListener {
// the media player
Player player = null;
// Component where video will appear
Component visualComponent = null;
/**
* Read the applet file parameter and create the media player.
*/
public void init() {
String mediaFile = null;
URL mediaURL = null;
setLayout(new BorderLayout());
if ((mediaFile = getParameter("file")) == null) {
System.err.print("Media File not present.");
System.err.println("Required parameter is 'file'");
}
else {
try {
mediaURL = new URL(getDocumentBase(), mediaFile);
player = Manager.createPlayer(mediaURL);
if(player != null) {
```

```
//tell the player to add this applet as a listener
player.addControllerListener(this);
}
else
System.err.println(“failed to create player for “ + mediaURL);
}
catch (IOException e) {
System.err.println(“URL for “+mediaFile+” is invalid”);
}
catch(NoPlayerException e) {
System.err.println(“Could not find a player for “ + mediaURL);
}
}
}
public void start() {
if (player != null)
//prefetch starts the player.
//Prefetch returns immediately, just like getImage
player.prefetch();
}
public void stop() {
if (player != null) {
// Stop media playback
player.stop();
//release resources for the media
player.deallocate();
}
}
//------ Controller interface method ----------
/**
* Whenever there is a media event,
* the controllerUpdate method is called
* for all the Player’s listeners
*/
public synchronized void controllerUpdate(ControllerEvent
event) {
if (event instanceof RealizeCompleteEvent) {
// Once the player has been realized add
// the visual component to the screen
if ((visualComponent = player.getVisualComponent()) != null)
add(“Center”, visualComponent);
// draw the component
validate();
}
```

```
else if (event instanceof PrefetchCompleteEvent) {
// start the player once it’s been prefetched
player.start();
}
}
}
```

Compiling the BasicPlayer :

To compile the BasicPlayer under JDK 1.2, simply use the javac compiler. However, underJDK1.1 it is also possible to compile the BasicPlayer. To do so, you need to first obtain a copy of the JMF classes from the vendor for your particular machine. For Windows 95/98, Windows NT, and Solaris users, you can obtain a set of classes from Sun Microsystems, Inc. athttp://java.sun.com/products/java-media/jmf/1.0/.

After you have downloaded the files from these sites and installed them, you need to track on the media classes. In the case of the Sun installation, the file you want is called jmf.jar and is installed in the plug-ins directory (if you’re using Netscape Navigator).To compile the program, you need to include the jmf.jar file in your classpath and then run the javac program.

Running BasicPlayer :

Before you can run the BasicPlayer, you must first create an HTML file with the appropriate file. In this case, we will use the sample MPEG file included with the JMF classes. Your HTML file should look similar to Listing 13.8.

Listing 13.8 BasicPlayer.html—A Basic HTML File That Includes Your Applet

```
<html>
<body>
<applet code=”BasicPlayer” width=”320" height=”300">
<param name=”FILE” value=”sample2.mpg”>
Sorry, your browser does not support Java(TM) Applets.
</applet>
</body>
</html>
```

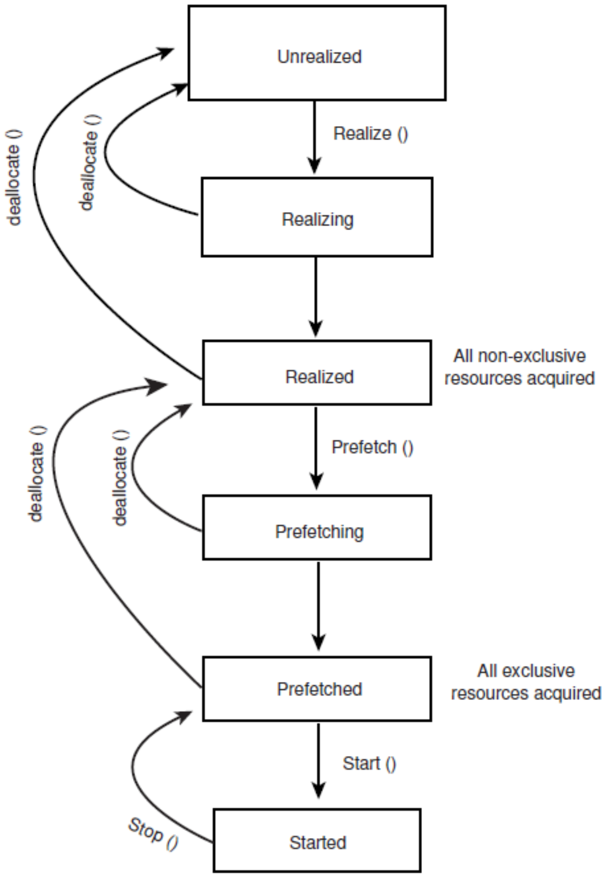
You now have two options for running BasicPlayer. First, you can use Netscape 4.03 with the final JDK 1.1 patch, or Internet Explorer 4, and if you have properly installed the media files from your vendor, you can now open the BasicPlayer.html file. The second option is to use the Appletviewer program included with the JDK. To use Appletviewer, first make sure that you still have the media.zip file included in your classpath, and then simply run it the way you are used to: Appletviewer BasicPlayer.html

If everything has gone as planned, you should see and hear the promotional Intel video start to play eventually.
Creating a Media Player

13.3 THE STATES OF THE PLAYER

Now that you have made it through the most basic player, it's time to go back and take a look at the stages or states that the player goes through as it progresses. Figure 13.1 shows the states that a player goes through during normal operation.

FIG. 13.1
A player travels through many states before it is started.



From Figure 13.1, you can see why we needed to track the state of the player before each method is called. The states of the player are as listed here:

Unrealized: When you first create a player, it is in the unrealized state. At this point, the player does not know anything about the media except what the URL to the media is. To move the player to the next stage, you can use the realize() method.

The prefetch() method will cause the realize() method to be run if the player is not yet realized. Therefore, you can skip over the realize() method as we did in the BasicPlayer and jump straight to the prefetch() method if you want.

Realizing: In the realizing state, the player determines the resource requirements for the particular media. For instance, it might require a rendering engine to play a FLIC file. In the realizing state, the player acquires all of these resources that are non-exclusive. A non-exclusive resource is a resource that can be shared with multiple players. The exclusive resources are acquired later in the prefetching state.

Realized*:* When a player enters the realized state, the RealizeCompleteEvent is issued. In the realized state, the player knows all the resources it will need in order to render the media. In addition, it knows enough about the media itself to be able to present the visual component of the media. The realized player has a “connection” to all the resources it will need, but it does not actually own any of the resources that would prevent another player from starting. These resources are known as scarce resources because they cannot be shared among different programs within the computer.

Prefetching:To get the player to move into the prefetching state, you can use the prefetch() method. In the prefetching state, the player is preloading some of the media it is preparing to present. It also obtains those scarce resources it couldn’t obtain back in the realizing state. The start() method will cause the prefetch() method to be called on a player if the media has not been fetched yet. However, unlike prefetch(), start cannot be called on an unrealized player. Also note that the prefetching state might have to be reentered if the media is repositioned, or if the time base for the media is changed and requires additional amounts of the media to be downloaded and buffered.

Prefetched*:* Upon entering the prefetched state, a player issues the PrefetchCompleteEvent.When it is in the prefetched state, a player is ready to go. It has obtained all its resources and has enough of the media to begin playing. In short, it is ready to be started. *Started:* As you saw in the BasicPlayer, to get a player to start you need to call the start()method. When a player is started, it enters the started state. After a player has been started, its clock is running and its media time has been set. If the player is not waiting for a particular time to start playing, it will begin immediately; otherwise, it will await the appropriate time and begin.

13.4 ADDING CONTROLS TO THE PLAYER

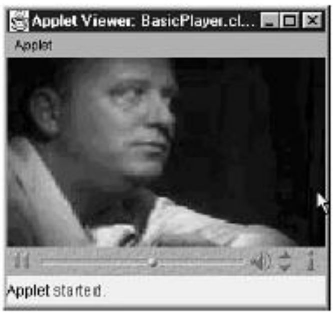
Now that you have the player in your applet, what will you do if you want to let the user control the motion of the player? The Media API has thought ahead for you. Because obviously you don’t want to be responsible for trying to figure out what controls are necessary for each type of player, the player has the capability to give you a set of controls using the getControlPanelComponent() method.

Like the getVisualComponent() method, the getControlPanelComponent() cannot be used until after the player has been realized. If you look back in the preceding section, you will see why. Until a player has been realized, it really doesn’t know what type of media it will be playing. Listing 13.9 shows a BasicPlayer with the control panel added. Figure 13.2 shows what it willlook like when you run BasicPlayer.Listing 13.9 Adding the Control Panel Is Simple and Easy in thecontrollerUpdate Method.

```
import java.applet.*;
import java.awt.*;
import java.net.*;
import java.io.*;
import javax.media.*;
/**
* A basic media player Applet
*/
public class BasicPlayer extends Applet implements ControllerListener {
Player player = null; // the media player
Component visualComponent = null; // Component where video will appear
Component controlComponent = null;
/**
* Read the applet file parameter and create the media player.
*/
public void init() {
String mediaFile = null;
URL mediaURL = null;
setLayout(new BorderLayout());
if ((mediaFile = getParameter(“file”)) == null) {
System.err.println(“Media File not present. Required parameter is
 ‘file’”);
}
else {
try {
mediaURL = new URL(getDocumentBase(), mediaFile);
player = Manager.createPlayer(mediaURL);
if(player != null) {
//tell the player to add this applet as a listener
player.addControllerListener(this);
}
else
System.err.println("failed to create player for " + mediaURL);
}
catch (IOException e) {
System.err.println("URL for "+mediaFile+" is invalid");
}
catch(NoPlayerException e) {
System.err.println("Could not find a player for " + mediaURL);
}
}
}
public void start() {
if (player != null)
//prefetch starts the player.
//Prefetch returns immediately, just like getImage
player.prefetch();
}
public void stop() {
if (player != null) {
// Stop media playback
player.stop();
//release resources for the media
player.deallocate();
}
}
//------ Controller interface method ----------
/**
* Whenever there is a media event, the
* controllerUpdate method is called
* for all the Player's listeners
*/
public synchronized void controllerUpdate(ControllerEvent
event) {
if (event instanceof RealizeCompleteEvent) {
// Once the player has been realized add
// the visual component to the screen
if ((visualComponent = player.getVisualComponent()) != null)
add("Center", visualComponent);
if ((controlComponent = player.getControlPanelComponent()) !=
null)
if(visualComponent != null)
add("South",controlComponent);
```

```
else
add(“Center”,controlComponent);
// draw the components
validate();
}
else if (event instanceof PrefetchCompleteEvent) {
// start the player once it’s been prefetched
player.start();
}
}
}
```

FIG. 13.2
The player has a default control panel.



Applet started.

13.5 CONTROLLING THE PLAYER PROGRAMMATICALLY

In addition to using the control panel, you can control the motion of the programmer using methods in player. You might want to do this for several reasons. One reason might be that you want to put up buttons to allow the player to control the player, but you don’t want to use the default control panel. Another reason is that you might want to change the way the player is playing because of some other condition in your program.

Starting the Player :

As you have already seen in the BasicPlayer example, player has a basic method—start(), which tells the player to start. However, there is a more fundamental method that allows you to start a player and specify when it will actually start displaying its media. The syncStart()method is the method that actually causes the player to start.

First, here’s a brief review of what the start() method does. Based on the state of the player, the start() method will sequentially call first prefetch() for all the players under its control(later in this chapter, controlling multiple players is discussed) and bring them into the prefetched state, and then will call syncStart().

Setting the Media Time :

Often you would like to take one video and jump around it. It’s not unlike playing tracks two and five on your CD player without playing through the whole CD. Fortunately, the media framework accommodates just such a need with the setMediaTime() method. The setMediaTime() method takes a long number as a parameter, and that number represents the time in nanoseconds. Unlike with most time increments you might be familiar with (such as when sleeping a thread), you are not dealing in milliseconds here, but the finer nanosecond increment.

Changing the Rate of Play:

Have you ever wanted to speed up or slow down video? How about play it backward? The player has a method for just such an activity. The method is setRate(), and it allows you to change the rate at which the media is played. However, you can use setRate() only before the player is actually started. Calling setRate on a started player causes an exception to be thrown.

The setRate() method might not be able to actually set the rate of the media play. You see, the only thing that’s guaranteed is that a player will be able to play the media at a rate of 1.0. The setRate() method returns to you the actual rate that has been applied. Fortunately, the AVI player supports playing the file at many different rates, so you can add a rate change to the BasicPlayer. Change the controllerUpdate method in BasicPlayer as shown in Listing 13.10.

Listing 13.10 Change the controllerUpdate Method to Adjust the Rate of Play

```
public synchronized void controllerUpdate(ControllerEvent event)
{
if (event instanceof RealizeCompleteEvent)
{
// Once the player has been realized add the visual component to the screen
if ((visualComponent = player.getVisualComponent()) != null)
add(“Center”, visualComponent);
if ((controlComponent = player.getControlPanelComponent()) != null)
if(visualComponent != null)
add(“South”,controlComponent);
else
add(“Center”,controlComponent);
// draw the components
validate();
}
else if (event instanceof PrefetchCompleteEvent)
{
System.out.println(“prefetching:”+(new Date()));
player.setRate((float)2.0);
// start the player once it’s been prefetched
player.start();
}
}
```

Changing the Sound Volume :

The capability to change the volume of the sound for the player is very important. To change the volume, the JMF has dedicated a whole class: GainControl. The first step in changing the volume is to get the GainControl from the player. To do so, you can invoke the getGainControl() method. Some players do not have an audio component to them. Those players return null from the getGainControl method, but for all others, after you have the GainControl, you can change the volume in many ways. Controlling the Player Programmatically Perhaps the most obvious thing to do with the audio is to turn it off all together, or mute it. Loand behold, the GainControl class just happens to have a mute() method for this purpose. Aside from muting the volume altogether, there are two distinctly different ways to control the volume of the player. The first way is to adjust the volume based on a level. A level is an arbitrary number between 0 and 1.0. When set to 1.0, the volume is basically on full. Set to 0, it’s almost indistinguishable from the muted setting. To change the GainControl’s volume level, you can use the setLevel() method. There is also an accompanying getLevel() method to retrieve the level of the GainControl. Remember that the only valid numbers are between 0 and 1.0.The problem with the level mechanism, however, is that it doesn’t allow you as a programmer to know how loud the media will actually play. Typically, sound engineers deal with sound in terms of decibels. The level only tells you that if you set the level to 0.5, it’s half as loud as at1.0. Fortunately, the media API provides a means to set the decibel level of the volume. However, this technique is really only a guess on the media player’s standpoint. It cannot account for things like the volume of your external speakers, but it instead makes a reasonable guess based on your system.

Resizing a Media Player:

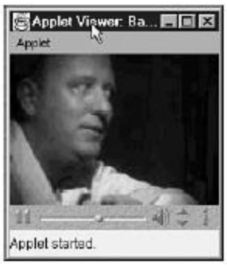
The media framework has the capability to allow you to resize a media component. Obviously, you want to be able to add a media player’s visual component to a panel just like any other component. To let the LayoutManager do its work, it’s necessary for the media player to allow you to resize the component. To see this at work, we need do no more than change the WIDTH and HEIGHT parameters in the HTML file for the BasicPlayer. Because the border layout will automatically resize the visual component to fit, if you shrink the size of the applet, you will witness the automatic rescaling of the media component. Try using the HTML file as shown in Listing 13.11. Figure 13.3 demonstrates just what BasicPlayer will look like now.

Listing 13.11 BasicPlayer2.html—Reducing the width and height Parameters

Helps Show Rescaling

```
<html>
<body>
<applet code=”BasicPlayer” WIDTH=”160" HEIGHT=”150">
<param name=”FILE” value=”ExampleVideo.avi”>
Sorry, your browser does not support Java(TM) Applets.
</applet>
</body>
</html>
```

FIG. 13.3
The media framework automatically rescales to the applet’s size.



Applet started.

Adding a Progress Bar:

As you know, downloading a multimegabyte file over the Internet can take a long time. In fact, it might seem an eternity during the time it takes to download a file. If the download takes too long, users are even likely to think that the whole system has hung. However, if there is some activity from your applet, a user will realize that the system itself has not hung. The file won’t take any less time to download, but the users might be a bit more tolerant. One perfect way to establish some activity is with a progress bar. As you will recall, ImageObservers are notified each time more of the image is downloaded off of the Internet. Likewise, a media player generates an event, CachingControlEvent; underlying this is the CachingControl.

Most players have a controller whose sole responsibility is to know the status of the download of the media. In addition, fortunately for you, the CachingControlhas a progress-bar component. Much like the visual component and the control-component elements that the player has, the progress-bar component can be added to any container. We can use the progress bar to add a great deal to the user's experience. Listing 13. 13 shows the BasicPlayer with the progress bar added to it.

Listing 13. 13 ProgressPlayer.java—The BasicPlayer with a Progress BarAdded

```
import java.applet.*;
import java.awt.*;
import java.net.*;
import javax.media.*;
import java.util.Date;
/**
* A basic media player Applet
*/
public class ProgressPlayer extends Applet implements ControllerListener
{
Player player = null; // the media player
Component visualComponent = null; // Component where video will appear
Component controlComponent = null;
Component progressBar = null;
/**
FIG.  13.3
The media framework automatically rescales to the applet's size.
* Read the applet file parameter and create the media player.
*/
public void init()
{
System.out.println("init:"+(new Date()));
String mediaFile = null;
URL mediaURL = null;
setLayout(new BorderLayout());
if ((mediaFile = getParameter("file")) == null)
{
System.err.println("Media File not present.
  Required parameter is 'file'");
}
else
{
```

```
try
{
mediaURL = new URL(getDocumentBase(), mediaFile);
player = Manager.createPlayer(mediaURL);
if(player != null)
{
//tell the player to add this applet as a listener
player.addControllerListener(this);
}
else
System.err.println(“failed to creat player for “ + mediaURL);
}
catch (MalformedURLException e)
{
System.err.println(“URL for “+mediaFile+” is invalid”);
}
catch(NoPlayerException e)
{
System.err.println(“Could not find a player for “ + mediaURL);
}
}
}
public void start()
{
if (player != null)
//prefetch starts the player.
//Prefetch returns immediately, just like getImage
player.prefetch();
}
public void stop()
{
if (player != null)


{
// Stop media playback
player.stop();
//release resources for the media
player.deallocate();
}
}
//------ Controller interface method ----------
/**
```

```
* Whenever there is a media event, the controllerUpdate method is called
* for all the Player’s listeners
*/
public synchronized void controllerUpdate(ControllerEvent event)
{
if (event instanceof RealizeCompleteEvent)
{
// Once the player has been realized add the
 visual component to the screen
if ((visualComponent = player.getVisualComponent()) != null)
add(“Center”, visualComponent);
if ((controlComponent = player.getControlPanelComponent()) != null)
if(visualComponent != null)
add(“South”,controlComponent);
else
add(“Center”,controlComponent);
// draw the components
validate();
}
else if (event instanceof PrefetchCompleteEvent)
{
System.out.println(“prefetching:”+(new Date()));
// start the player once it’s been prefetched
player.start();
}
else if (event instanceof CachingControlEvent) {
// Put a progress bar up when downloading starts,
// take it down when downloading ends.
CachingControlEvent cce = (CachingControlEvent) event;
CachingControl cc = cce.getCachingControl();
long cc_progress = cce.getContentProgress();
long cc_length = cc.getContentLength();
if (progressBar == null) // Add the bar if not already there ...
if ((progressBar = cc.getProgressBarComponent()) != null) {
add(“North”, progressBar);
validate();
}

if (progressBar != null) // Remove bar when finished downloading
if (cc_progress == cc_length) {
```

```
remove (progressBar);
progressBar = null;
validate();
}
}
}
}
```

FIG. 13.4

While the player downloads the media, the progress bar shows the status.

Applet started.

13.6 LINKING MULTIPLE PLAYERS

When you have more than one player working on a panel, it’s very likely that you would like to start and stop them with a single controller. To do this, you need to be able to link the controllers. The player has a method for just this purpose: addController(). The purpose of the addController method is to tell the player to also control the other player. To review this means that with the linemasterPlayer.addController(slavePlayer);

slavePlayer is controlled by masterPlayer. However, before you can control another player, the slave player needs to be realized. The reason for this goes back to how much the player knows about itself when it’s first created. As you will recall, before a player is actually realized, it doesn’t even know what kind of media it will be playing. At that point, it doesn’t know what kind of controller it will have, so it can’t give control to another player. To gain control of another player, the time bases for both players must be compatible. To make sure that this is the case, the master player will try to give the slave player its time base. If this attempt fails, however, the addController method will throw anIncompatibleTimeBaseException.

Listing 13.13 is an example of the BasicPlayer with two side-to-side players, which are controlled with a single control.

After the player starts, you will see a single control panel. Press the play button and they will both start.

Listing 13.13 The Primary Player

```
import java.applet.*;
import java.awt.*;
import java.net.*;
import javax.media.*;
import java.util.Date;
/**
* A basic media player Applet
*/
public class PrimaryPlayer extends Applet implements ControllerListener
{
Player player = null; // the media player
Player slavePlayer = null;
Component visualComponent = null; // Component where video will appear
Component slaveVisualComponent = null;
Component controlComponent = null;
Component progressBar = null;
/**
* Read the applet file parameter and create the media player.
*/
public void init()
{
String mediaFile = null;
URL mediaURL = null;
setLayout(new BorderLayout());
if ((mediaFile = getParameter(“file”)) == null)
{
System.err.println(“Media File not present.
  Required parameter is ‘file’”);
}
else
{
try
{
mediaURL = new URL(getDocumentBase(), mediaFile);
player = Manager.createPlayer(mediaURL);
slavePlayer = Manager.createPlayer(mediaURL);
if(player != null)
{
```

```
//tell the player to add this applet as a listener
player.addControllerListener(this);
}
else{
System.err.println(“failed to create player for “ + mediaURL);
}
if(slavePlayer != null)
{
//tell the player to add this applet as a listener
slavePlayer.addControllerListener(this);
}
else{
System.err.println(“failed to create player for “ + mediaURL);
}
}
catch (MalformedURLException e)
{
System.err.println(“URL for “+mediaFile+” is invalid”);
}
catch(NoPlayerException e)
{
System.err.println(“Could not find a player for “ + mediaURL);
}
}
}
public void start()
{
if (player != null)
//prefetch starts the player.
//Prefetch returns immediately, just like getImage
player.prefetch();
if (slavePlayer !=null)
slavePlayer.prefetch();
}
public void stop()
{
if (player != null)
{
// Stop media playback
player.stop();
//release resources for the media
player.deallocate();
}
```

```
}
//------ Controller interface method ----------
/**
* Whenever there is a media event, the controllerUpdate method is called
* for all the Player’s listeners
*/
public synchronized void controllerUpdate(ControllerEvent event)
{
if (event instanceof RealizeCompleteEvent)
{
System.out.println(“realized”);
// Once the player has been realized add the visual
  component to the screen
if (event.getSource() == slavePlayer){
System.out.println(“slave”);
if ((slaveVisualComponent = slavePlayer.
 getVisualComponent()) != null)
add(“West”, slaveVisualComponent);
try{
player.addController (slavePlayer);
}
catch (IncompatibleTimeBaseException e)
{
System.err.println(“Could not attach player “+e);
}
validate();
}
else{
if ((visualComponent = player.getVisualComponent()) != null)
add(“East”, visualComponent);
if ((controlComponent = player.getControlPanelComponent()) != null)
if(visualComponent != null)
add(“South”,controlComponent);
else
add(“Center”,controlComponent);
// draw the components
validate();
}
}
if (event instanceof PrefetchCompleteEvent){
//player.setRate((float)2.0);
// start the player once it’s been prefetched
//player.setMediaTime(20000);
player.start();
}
else if (event instanceof CachingControlEvent) {
// Put a progress bar up when downloading starts,
// take it down when downloading ends.
CachingControlEvent e = (CachingControlEvent) event;
CachingControl cc = e.getCachingControl();
long cc_progress = e.getContentProgress();
long cc_length = cc.getContentLength();
if (progressBar == null) // Add the bar if not already there ...
if ((progressBar = cc.getProgressBarComponent()) != null) {
add(“North”, progressBar);
validate();
}
if (progressBar != null) // Remove bar when finished downloading
if (cc_progress == cc_length) {
remove (progressBar);
progressBar = null;
validate();
}
}
}
}
```

13.7 CREATING YOUR OWN MEDIA STREAM

Pull Media Streams:

As you are aware, there are several ways to get data into a client. And more important, there are two basic subsets for retrieving data. One basic type is pull media streams, in which the client requests that the server send it something. This is how your Web browser works. You go out to a Web site and click on a URL. Your browser asks the server to send it that Web page, and off things go. The other type of data stream, which we will talk about later, is called push media. Push media results when the server automatically sends something to the client, or as some have put it, broadcasts information to your client.

Pull media is typical of many of the Internet protocols. For instance, the two most popular protocols on the Internet, HTTP and FTP, are both examples of pull streams. You have already seen how the HTTP protocol can be utilized in the BasicPlayer. Next, you’ll explore how to build your own pull stream.

Push Media Streams :

Push streams have recently been popularized by Pointcast and Marimba. One basic advantage of pull streams is that you can guarantee that the client will receive 100 percent of all the data sent from the server. Because of this, your player does not need to know how to accommodate the gaps in data that push streams are likely to have. The push player needs to know how to handle gaps in data when they accrue, and it needs to be able to account for them.

13.8 A LARGER APPLICATION

Now that you have read about all the controls for the player, you can put everything together and create your own custom control panel. You'll use all the methods you've learned to use up until now.

Listing 13.14 CustomPlayer.java

```
import java.applet.*;
import java.awt.*;
import java.net.*;
import java.media.*;
/**
* This is a Java Applet that demonstrates how to add your own
* custom controls to the basic media player.
*/
public class ExtendedPlayer3 extends Applet implements
ControllerListener
{
Player player = null; // media player
Component visualComponent = null;
// component in which the video is playing
boolean running = false; // indicates if the applet
  is currently running, because the user is on the page
/**
* The component of the media player that holds the gainControl
reference.
*/
GainControl gainControl = null;
/**
* Panel used to hold the custom controls in the Applets Layout
Manager.
*/
Panel controlPanel = null;
/**
```

```
* Runnable class used to monitor media progress and update
the UI.
*/
MediaProgressMonitor progressMonitor = null;
/**
* Buttons within the custom controls used to adjust Starting,
Stopping,
* Gain increase and Gain decrease.
*/
Button startButton = null;
Button stopButton = null;
Button gainUpButton = null;
Button gainDownButton = null;
/**
* Labels within the custom controls used to indicate adjustable
media player
* features.
*/
Label controlLabel = null;
Label gainLabel = null;
Label muteLabel = null;
Label mediaTimesLabel = null;
/**
* Checkbox within the custom controls used control the Mute
feature of the
* media player.
*/
Checkbox muteCheckbox = null;
/**
Textfield within the custom controls used to indicate
  the current media player
* position and total media file duration.
*/
TextField timeText = null;
/**
* Read the applet file parameter and create the media player.
*/
public void init()
{
String mediaFile = null; // input filename from Applet parameter
URL mediaURL = null; // base URL for the
 document containing the applet
setLayout(new BorderLayout());
/**
* Get the media filename info.
* The applet tag should contain the path to the
* source media file, relative to the applet.
*/
if ((mediaFile = getParameter(“MediaFile”)) == null)
{
System.err.println(“Invalid media file parameter”);
}
else
{
try
{
// Create an url from the file name and the url to the document
containing this applet.
mediaURL = new URL(getDocumentBase(), mediaFile);
// Create an instance of an appropriate
 media player for this media type.
player = Manager.createPlayer(mediaURL);
if(player != null)
{
// Add this applet as a listener for the media player events
player.addControllerListener(this);
// Create the duration monitor object:
progressMonitor = new MediaProgressMonitor(this);
}
else
System.err.println(“Could not create player for “ + mediaURL);
}
catch (MalformedURLException e)
{
System.err.println(“Invalid media file URL!”);
}
catch(NoPlayerException e)
{
System.err.println(“Could not find a player to
  create for” + mediaURL);
}
}
}
/**
* Start media file playback. This method is called the first time
* that the Applet runs and every time the user re-enters the
page.
*
```

```
* Call prefetch() to prepare to start the player. Prefetch returns
* immediately, so this method does not call player.start(). The
* controllerUpdate() method will call player.start() once the
* player is Prefetched.
*/
public synchronized void start()
{
if (player != null)
{
player.prefetch();
running = true;
}
}
/**
* Stop media file playback and release resources before leaving
* the page.
*/
public synchronized void stop()
{
if (player != null)
{
progressMonitor.stop();
player.stop();
player.deallocate();
running = false;
}
}
/**
* This controllerUpdate method must be defined in order to implement
* a ControllerListener interface. This method will be called whenever
* there is a media event.
*/
public synchronized void controllerUpdate(ControllerEvent event)
{
// do nothing if player is set to null
if (player == null)
return;
// When the player is Realized, get the visual component
// and control component and add them to the Applet
if (event instanceof RealizeCompleteEvent)
{
```

```
if ((visualComponent = player.getVisualComponent()) != null)
add(“Center”, visualComponent);
// Get pointer to the Gain Control of the media player.
gainControl = player.getGainControl();
// Create the custom control components.
createCustomControls();
if (visualComponent != null)
add(“South”,controlPanel);
else
add(“Center”,controlPanel);
// force the applet to draw the components
validate();
}
// Once the player has Prefetched, start it
else if (event instanceof PrefetchCompleteEvent)
{
if(running)
{
player.start();
progressMonitor.start();
}
}
// If we’ve reached the end of the media “rewind” it to the
beginning.
else if (event instanceof EndOfMediaEvent)
{
player.setMediaTime(0);
if (running)
player.start();
}
// A fatal player error has occurred
else if (event instanceof ControllerErrorEvent)
{
progressMonitor.stop();
player = null;
System.err.println(“FATAL ERROR: “ +
 ((ControllerErrorEvent)event).getMessage());
}
}
/**
* This method handles the AWT details required to display
* player controls.
*/
```

```
public void createCustomControls()
{
controlPanel = new Panel();
GridBagLayout gridBag = new GridBagLayout();
GridBagConstraints gridBagCon = new GridBagConstraints();
controlPanel.setFont(new Font(“Arial”, Font.PLAIN, 14));
controlPanel.setLayout(gridBag);
// Create the first row of AWT control components:
 Label, Start Button, Stop Button.
gridBagCon.fill = GridBagConstraints.BOTH;
gridBagCon.weightx = 1.0;
controlLabel = new Label(“Controls:”, Label.LEFT);
makeControl(controlPanel, controlLabel, gridBag, gridBagCon);
startButton = new Button(“Start”);
makeControl(controlPanel, startButton, gridBag, gridBagCon);
gridBagCon.gridwidth = GridBagConstraints.REMAINDER;
stopButton = new Button(“Stop”);
makeControl(controlPanel, stopButton, gridBag, gridBagCon);
// Create the second row of AWT control components:
 Label, GainUp Button, GainDown Button.
gridBagCon.weightx = 1.0;
gridBagCon.gridwidth = 1;
gainLabel = new Label(“Gain:”, Label.LEFT);
makeControl(controlPanel, gainLabel, gridBag, gridBagCon);
gainUpButton = new Button(“Loud”);
makeControl(controlPanel, gainUpButton, gridBag,
gridBagCon);
gridBagCon.gridwidth = GridBagConstraints.REMAINDER;
gainDownButton = new Button(“Soft”);
makeControl(controlPanel, gainDownButton, gridBag,
gridBagCon);
// Create the third row of AWT control components: Label, Mute
checkbox.
gridBagCon.gridwidth = GridBagConstraints.RELATIVE;
gridBagCon.weightx = 1.0;
muteLabel = new Label(“Mute:”, Label.LEFT);
makeControl(controlPanel, muteLabel, gridBag, gridBagCon);
gridBagCon.gridwidth = GridBagConstraints.REMAINDER;
muteCheckbox = new Checkbox(“”);
makeControl(controlPanel, muteCheckbox, gridBag,
gridBagCon);
// Create the third row of AWT control components:
 Label, media time textbox.
gridBagCon.gridwidth = 1;
gridBagCon.weightx = 1.0;
mediaTimesLabel = new Label("Current Time/Total Time:",
Label.LEFT);
makeControl(controlPanel, mediaTimesLabel, gridBag,
gridBagCon);
gridBagCon.gridwidth = GridBagConstraints.REMAINDER;
timeText = new TextField("0.0//0.0", 13);
timeText.setEditable(false);
makeControl(controlPanel, timeText, gridBag, gridBagCon);
}
/**
* This method adds a control to the custom control layout
manager.
*/
protected void makeControl(Container parentComp, Component
 newComponent, GridBagLayout gridbag, GridBagConstraints
constraint)
{
gridbag.setConstraints(newComponent, constraint);
parentComp.add(newComponent);
}
/**
* This method captures all the events from the custom controls.
This
* is where each control calls the media player control methods.
*/
public boolean action(Event evt, Object arg)
{
if (evt.target instanceof Button)
{
// Process the button event:
if ("Start".equals(arg))
{
player.start();
}
else if ("Stop".equals(arg))
{
player.stop();
}
else if ("Loud".equals(arg))
{
gainControl.setDB(2.0f);
}
else if ("Soft".equals(arg))
{
gainControl.setDB(-2.0f);
}
return(true);
}
else if (evt.target instanceof Checkbox)
{
// Set the player’s Mute control based upon the checkbox state.
if (muteCheckbox.getState() == true)
gainControl.setMute(true);
else
gainControl.setMute(false);
return(true);
}
else
return(false);
}
}
/**
* This class is used to continually monitor the progress
* of the playing media file. The thread wakes up every 50 millisec
* and passes the progress info to the player controls in the applet.
*/
class MediaProgressMonitor implements Runnable
{
ExtendedPlayer3 m_Applet = null;
Thread thread = null;
boolean running;
public MediaProgressMonitor(ExtendedPlayer3 applet)
{
m_Applet = applet;
}
/**
* This method is called when the user starts the Applet or returns
* from another page.
*/
public synchronized void start()
{
thread = new Thread(this);
thread.start();
running = true;
```

```
}
/**
* This method is called when the user stops the Applet or
* leaves the page.
*/
public synchronized void stop()
{
running = false;
}
/**
* This method is called after the start method has executed.
* Every 50 milliseconds, check the media player’s progress
* and forward the results to the player’s control component.
*/
public void run()
{
String mediaDuration = null;
String mediaTime = null;
char tmpChar;
// Get the total time of the media file and store for use later.
long duration = m_Applet.player.getDuration();
mediaDuration = new String(String.valueOf(duration / (long) 1e08));
tmpChar = mediaDuration.charAt(mediaDuration.length() - 1);
mediaDuration = String.valueOf(duration / (long) 1e09) + “.” + tmpChar;
while (running)
{
// Update the media time text field.
long currtime = m_Applet.player.getMediaTime();
mediaTime = new String(String.valueOf(currtime / (long) 1e08));
tmpChar = mediaTime.charAt(mediaTime.length() - 1);
mediaTime = String.valueOf(currtime / (long) 1e09) + “.” + tmpChar;
m_Applet.timeText.setText(mediaTime + “/” + mediaDuration);
try
{
thread.sleep(50);
}
catch (InterruptedException e)
{
System.err.println(e);
}
}
}
}
```

Security Support with the JCC :

As the number of users on the Internet grows, more and more companies want to sell products and services online. One of the biggest hurdles for online sales is security. Although the Java Security API provides the strong encryption needed to safely pass credit card and banking information back and forth over the Internet, there is still a need for a common set of APIs for performing electronic transactions. The Java Commerce Client (JCC) was designed to support all the features needed for online transactions.

The JCC includes a standard set of user interfaces, a framework for passing messages between clients and servers, and set of APIs for handling various aspects of online transactions. One of the core features of the JCC is the *cassette*, which is a collection of classes performing particular operation. You can think of a cassette as a larger version of an object—it consists of multiple objects and some extra information, but it “plugs in” to the commerce framework to perform some desired function. The Java Commerce Client was originally called the Java Electronic Commerce Framework(JECF). You will probably still see it called JECF in many places, including Sun’s documentation. n

Commerce Messages:

In a client/server environment, it is important to nail down a protocol early so that development can proceed on both the client and server sides. JCC defines a message format called Java Commerce Messages (JCM) to make it easy to define interactions between clients and servers.

JCM messages are human-readable text messages containing multiple name=value pairs likethis:buyer.billto.name = Mark Wutka

The contents of the message will vary from server to server. Typically, the message will contain information about the buyer, the order, the protocol, and any additional requirements.JECF also defines a MIME type for JCMs so that e-mail programs, web servers, browsers, and other MIME-aware applications can deal with them. The MIME type is application/x-javacommerce,and a JCM has a file extension of .jcm.

Typically, the server will determine what kind of data needs to be sent to perform a transaction. Chances are, there will be a core set of name=value pairs that every server expects, and additional server-specific extensions that vary according to the type of server.

Creating Cassettes :

Cassettes are the core of JCC. They perform all the operations needed to get information from the user, validate the transaction, and send it to a server. Cassettes are grouped into distinct areas indicating what type of operation they perform. The types of cassettes are instruments, protocols, operations, services, and user interfaces (UI).An instrument cassette represents data used in a transaction. A typical instrument might represent a credit card, containing the credit card type, number, and expiration date. Another instrument might represent a bank account, containing the bank location, the account number, and the type of account.

A protocol is a communications mechanism between JCC and a commerce server. You have probably encountered the term “protocol” in reference to networking, where you find the File Transfer Protocol (FTP) and the Hypertext Transfer Protocol (HTTP). In the world of electronic commerce, you find protocols such as SET, which is a standard for secure electronic transactions. Although SET defines the interactions between a client and a server, the protocol cassette actually maps various protocols required by operation cassettes onto communication protocols. JCC has a protocol defined for making a purchase, for example. A protocol cassette for the SET protocol would implement the interface for the purchase protocol and perform purchases using the SET protocol. An operation cassette represents a task that a user may want to perform. A very common operation cassette is one that implements a purchase operation, enabling the user to make a purchase. You might also need a sell operation enabling the user to sell something electronically. Most of the time, selling is performed on a server; cases may arise, however, where the user is doing the selling—as in a stock trade.

A UI cassette presents the user interface for performing various operations. An ATM UI might present all the operations found at an automatic teller machine, for example, although dispensing cash might be a little tricky because paper doesn’t travel well over a modem and stores are hesitant to accept money that has been faxed. Still, you should be able to transfer money between accounts and possibly recharge a debit card.

A service cassette is a “helper” cassette that doesn’t necessarily perform a transaction itself. A common service cassette might be a Rolodex of credit cards or a visual stock portfolio. A service cassette can also provide services to other cassettes. An operation cassette, for example, may use a service cassette to present a user interface when the operation is being performed.

The CassetteControl Class:

Each cassette contains a class named CassetteControl, which contains information about the contents of a cassette, the kind of function it performs, the current version, and any cassettes it may depend on. The CassetteControl class is used when the cassette is first installed into theJCC environment, and also at runtime when the version is checked.

Creating Cassettes:

When you create your CassetteControl class, you must declare it as a subclass of Cassette. There is no CassetteControl base class. The Cassette class loader specifically looks for a class named CassetteControl and expects it to be a subclass of Cassette. One of the helper classes that you will need when you create a CassetteControl object is theCassetteIdentifier, which contains the name and version information of a cassette. You can create a CassetteIdentifier one of three ways:

```
CassetteIdentifier()
CassetteIdentifier(String name)
CassetteIdentifier(String name, int majorVersion,int minorVersion)
```

If you create an empty cassette identifier using the null (empty) Constructor, you must setthe name and version numbers with the setName, setMajorVersion, and setMinorVersionmethods:

```
public void setName(String name)
public void setMajorVersion(int majorVersion)
public void setMinorVersion(int minorVersion)
```

If you set the cassette identifier using only the name, you must embed the version information in the name by using this form:name_majorVersion.minorVersion

The Constructors of the following two cassette identifiers create identical identifiers, for example:

```
CassetteIdentifier("StockTrader", 3, 1)
CassetteIdentifier("StockTrader_3.1")
```

You use a cassette identifier in a ControlCassette object to provide the current version number of the cassette, as well as the identifiers for any other cassettes you may depend on. Each ControlCassette object must return its current version in thegetCurrentVersionIdentifier method. The following code fragment creates a version identifier on-the-fly:

```
public CassetteIdentifier getCurrentVersionIdentifier()
{
```

```
return new CassetteIdentifier(“StockTrader”, 3, 1);
}
```

It is more efficient, of course, to create a CassetteIdentifier ahead of time and just return it every time getCurrentVersionIdentifier is called. You also use a cassette identifier when returning the cassettes that your cassette depends on. The getDependencyIdentifiers method should return an array of cassette identifiers or null if there are no dependencies. Suppose, for example, that your cassette depends on cassettes named TradeOMatic and LeatherPortfolio. Your getDependencyIdentifiers method might look like this:

```
CassetteIdentifier[] getDependencyIdentifiers()
{
CassetteIdentifier dependencies[] =
new CassetteIdentifier[2];
dependencies[0] = new CassetteIdentifier(
“TradeOMatic”, 2, 1);
dependencies[1] = new CassetteIdentifier(
“LeatherPortfolio_2.4”);
return dependencies;
}
```

The getJCMForLatestVersion method should return an array of URLs telling JCC where to find newer versions of the cassette:

```
public URL[] getJCMForLatestVersion()
```

If another cassette needs a newer version of your cassette, the JCC will automatically check these URLs and download the needed version.

The install method in CassetteControl should register itself with JCC using one of the following methods:

```
public final void registerInstrumentType(String instrumentType, String className)
public final void registerProtocol(String protocolName, String className)
public final void registerOperation(String operationName, String className)
public final void registerService(String serviceName, String className)
public final void registerWalletUI(String walletUIName, String description,String className)
```

An install method for a StockTrader operation might look like this:

```
public void install()
throws CassetteInstallationException{
registerOperation(“StockTrader”,
“stocks.trader.cassettes.StockTrader”);
}
```

When a cassette is removed from the system, JCC calls the uninstall method. If your cassette

doesn’t need to do anything to uninstall, just create an empty method like this:

```
public void uninstall()
{}
```

Creating Cassettes:

When a cassette is started, its init method is called. When the cassette is stopped, its shutdown method is called:

```
public void init()
public void shutdown()
```

For the CassetteControl class, however, you don’t need the init and shutdown methods, sojust make them empty.The getExpirationDate method returns the date when the cassette expires:

```
public Date getExpirationDate()
```

When the system needs to update a cassette, it first asks the cassette whether it is okay toupdate it. The doUpdate method is passed the date of the last update and should return true ifit is okay to update, or false if it should not be updated:

```
public boolean doUpdate(Date lastUpdate)
```

The Instrument Cassette Class:

An instrument, in the JCC world, represents a source of data. Usually, an instrument represents something such as a credit card, a bank account, or even a stock. In a typical scenario, a user visiting an online store would decide to make a purchase. The JCC applet would present a collection of instruments, perhaps even looking like a wallet, and the user would select an “instrument” for payment, such as a credit card. The applet would use the instrument alongwith an operation cassette and a protocol cassette to transmit the order to the online store. The operation cassette and protocol cassette would communicate with the credit instrument to get all the information needed to complete the sale—the account number, the expiration date, and so forth.

An instrument implements the Instrument interface, which is a high-level description of the interfaces that every instrument should support. The description is high level because the functions of different interfaces can vary so greatly that there would be no way to predict all the necessary methods to put into the interface. Besides, you don’t want a stock instrument containing methods to get the credit card number and expiration date. Instead, the various types of instruments have their separate interfaces that implement the Instrument interface. The JCC comes with a GenericCreditCard interface, for example, which implements the Instrument interface. As you might expect, the GenericCreditCard interface has methods to query the account number, expiration date, billing address, and cardholder’s name. The Instrument interface contains methods that indicate the type and general function of the instrument. The getDescription method returns a simple text description of the instrument, for example:

public String getDescription()

The getType method returns the type of instrument, which might be “Visa”, “MasterCard”, “Amex”, or “Discover”:

public String getType()

The getName method, on the other hand, returns the name of the instrument, such as Sir ChargeAlot:
public String getName()

The getContext method returns a string indicating how the instrument is used:
public String getContext()

The current JCC documentation gives “pay” and “accumulate” as examples of contexts. These will hopefully be standardized some time in the future to avoid possible conflicts of terminology.

The getVisualRepresentation method returns an AWT component that represents the instrument:
public Component getVisualRepresentation(CommerceContext context, Dimension dim)

The visual representation of an instrument allows for the kind of flashy representation that marketing folks love. The visual representation can display animations and even play audio clips. (For a credit card, I recommend the Eagles’ “Take It to the Limit.”)

The getSimpleGraphic method returns an image to use in various selection screens:
public Image getSimpleGraphics(CommerceContext context, Dimension dim)

The simple graphic is intended for container displays where the user selects from a group of instruments. A commerce applet might display a graphic of a wallet containing your credit cards. After you select a credit card, the applet would use the visual representation object for the rest of the transaction. The InstrumentAdministration interface contains methods involved with creating and maintaining specific instruments. Several of these methods are used just to get the graphics components used for editing the instrument. The getNewInstrumentUI method, for example, returns an AWT container used for creating a new instrument:

```
        public Container getNewInstrumentUI(DataStore
instStore,CommerceContext context, Dimension dim)
```

The getInstrumentEditUI, on the other hand, returns a container used to edit existing instruments:

```
        public Container getInstrumentEditUI(DataStore
instStore,CommerceContext context, Dimension dim)
```

The getInstrument method returns an instrument from the database:

```
        public Instrument getInstrument(DataStore instStore)
```

The DataStore object used in the getInstrument, and new/edit user interfaces provide a mechanism for storing and retrieving instruments from a database. A DataStore actually representsa database blob (Binary Large Object) in which instruments are stored using the Java serialization API.

The DataStore object has only three methods:

```
public boolean commit() throws IOException
public void setObject(Serializable obj) throws IOException
public Serializable getObject() throws IOException
```

The setObject method stores an object in a data store; the getObject method retrieves an object from a data store. The commit method saves any data store changes and returns true if successful. Because credit cards are one of the most prevalent forms of payment currently in use, the JCC includes a generic credit card interface that provides the kind of information commonly found on credit cards. This gives implementers of protocol and operation cassettes a base to work with so that they don't have to wait to see how other developers implement credit card instruments. If you create a credit card instrument, it should implement the GenericCreditCardinterface.

The GenericCreditCard interface contains get and set methods for the items found in almost all credit cards. The get/set methods are as follows:

```
public String getPAN(); // PAN = Primary Account Number
public void setPAN(String primaryAccountNumber);
public String getExpireDate();
public void setExpireDate(String expireDate);
public String getCardholderName();
public void setCardholderName(String name);
public AddressRecord getBillingAddress();
public void setBillingAddress(AddressRecord address);
```

In addition, the accept method should return true if the current transaction is permitted:

```
public boolean accept();
```

Listing 13.1 shows an example instrument from the JCC package from Sun.

Listing 13.1 Source Code to CCInstrument.java

```
/* @(#)CCInstrument.java 1.21 11/07/97

 Copyright (c) 1996 Sun Microsystems, Inc. All Rights Reserved.

 Permission to use, copy, modify, and distribute this software
 and its documentation for NON-COMMERCIAL or COMMERCIAL purposes and
 without fee is hereby granted.
 Please refer to the file http://java.sun.com/copy_trademarks.html
 for further important copyright and trademark information and to
 http://java.sun.com/licensing.html for further important licensing
 information for the Java (tm) Technology.

 SUN MAKES NO REPRESENTATIONS OR WARRANTIES
ABOUT THE SUITABILITY OF THE SOFTWARE, EITHER
EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO
THE IMPLIED WARRANTIES OF MERCHANTABILITY,
FITNESS FOR A PARTICULAR PURPOSE, OR NON-
INFRINGEMENT. SUN SHALL NOT BE LIABLE FOR ANY
DAMAGES SUFFERED BY LICENSEE AS A RESULT OF
USING, MODIFYING OR  DISTRIBUTING THIS SOFTWARE
OR ITS DERIVATIVES.

          THIS SOFTWARE IS NOT DESIGNED OR INTENDED
FOR USE OR RESALE AS ON-LINE CONTROL EQUIPMENT
IN HAZARDOUS ENVIRONMENTS REQUIRING FAIL-SAFE
PERFORMANCE, SUCH AS IN THE OPERATION OF
NUCLEAR FACILITIES, AIRCRAFT NAVIGATION OR
COMMUNICATION SYSTEMS, AIR TRAFFIC CONTROL,
DIRECT LIFE SUPPORT MACHINES, OR WEAPONS
SYSTEMS, IN WHICH THE FAILURE OF THE SOFTWARE
COULD LEAD DIRECTLY TO DEATH, PERSONAL INJURY,
OR SEVERE PHYSICAL OR ENVIRONMENTAL DAMAGE
("HIGH RISK ACTIVITIES"). SUN  SPECIFICALLY DISCLAIMS
```

```
ANY EXPRESS OR IMPLIED WARRANTY OF FITNESS FOR
HIGH RISK ACTIVITIES.

*/

package com.sun.commerce.gencc;
import javax.commerce.util.AddressRecord;
import java.awt.*;
import java.net.URL;
import javax.commerce.util.Money;
import java.io.*;
import javax.commerce.base.*;
import javax.commerce.gui.*;
import javax.commerce.gui.image.*;

/*
 A generic credit card object in the user's wallet.
 @author Daniel J. Guinan
 @author Java Commerce Team
 @version @(#)CCInstrument.java 1.21 11/07/97
*/
public class CCInstrument implements GenericCreditCard
{
// The data that is to be stored for this instrument
/** The Primary Account Number of this credit-card */
protected String PAN = null;
/** The expiration date of this credit-card */
protected String expireDate = null;
/** The description of this credit-card */
protected String description = null;
/** The name of the cardholder of this credit-card */
protected String cardholderName = null;
/** The billing address for this credit-card */
protected AddressRecord billingAddress = null;
/** The local alias (the name in the wallet) of this credit-card */
protected String localAlias = null;
/** The name of the image (taken from the /graphics directory in
* the cassette */
protected String imageName = null;
// Declare any variables that are NOT to be stored with the
// instrument as transient
/** We keep the image around so that we don't have to re-create
it */
private transient Image theImage=null;
```

```
/** This dummy frame is present to do image processing. AWT
1.1 requirement
*/
Creating Cassettes
private static transient Frame dummy=null;
/** Constructor
* @param desc The description of the card to find in the wallet
* @param cardno The primary account number (the credit-card
number)
* @param expire The expiration date of the credit-card
* @param holder The cardholder name
* @param addr The card's billing address
* @param gra The card's local alias
*/
public CCInstrument (String desc,
String cardno,
String expire,
String holder,
AddressRecord addr,
String gra,
String imagenm)
{
description = desc;
PAN = cardno;
expireDate = expire;
cardholderName = holder;
billingAddress = addr;
localAlias = gra;
imageName = imagenm;
}
/****************************************************************
* Instrument interface specific methods
****************************************************************/
/** Returns the description of the instrument
* @return description
*/
public String getDescription ()
{ return description; }
/** Sets the description of the instrument.
* @param x The description of the credit-card.
*/
public void setDescription (String x)
{ description = x; }
```

```
/** Returns the context that this credit-card is generally used within.
* @return always returns “pay” for generic credit cards
*/
public String getContext() { return “pay”; }
/** Sets the instrument type
* @return In the case of Generic Credit Card, it always returns the
* constant <tt>CCAdmin.Type_Name</tt>
*/
public String getType ()
{ return CCAdmin.TYPE_NAME; }
/**
* Returns the name the user associated with the specific instance
* of the credit card
* @return The local (wallet name) name of the credit-card.
*/
public String getName()
{ return localAlias;}
/**
* Set the user-defined name of this instance of the credit card.
* @param x The local alias (wallet name) name of the credit-card.
*/
public void setName(String x)
{ localAlias = x; }
/**
* This method retrieves the visual representation of this instrument.
* Since this method returns a Component, it can be active imagery
* (e.g. an animation), or other such thing.
*
* @param media The CommerceContext used to fetch imagery from
* the jecf, a file, a URL, or the cassette.
*
* @param dim A hint as to the size of the image required. Since
* proportions are important, we will not be returning
* an image exactly this size. Rather, we will ensure
* that our returned image is proportionally correct,
* yet fits within these dimensions.
*
```

```
* @return Component A component with branding imagery associated with
* this instrument.
*/
public Component getVisualRepresentation(CommerceContext media, Dimension dim)
{
try // We will do the simplest thing — wrap the simple imagery
{ // into a canvas and return it.
Image img = getSimpleGraphic(media, dim);
CWImage cimg = new CWImage(img,dim);
//cimg.waitForDimensions();
return cimg;
} catch(Exception e) { return new Label(“No Imagery”); }
}
/**
* This method retrieves the image that is associated with this
* instrument.
*
* @param media The CommerceContext used to fetch imagery from
* the jecf, a file, a URL, or the cassette.
*
* @param dim A hint as to the size of the image required. Since
* proportions are important, we will not be returning
* an image exactly this size. Rather, we will ensure
* that our returned image is proportionally correct,
* yet fits within these dimensions.
* @return Image The image representing this instrument.
*/
public Image getSimpleGraphic(CommerceContext media, Dimension dim)
{
try
```

```
{
Image img,timg;
if(dummy==null) // We need a dummy frame to do image processing
{ // create it if we haven’t already
dummy = new Frame();
dummy.addNotify(); // We need the Frame’s peer to exist for
} // this to work.. This forces that to happen.
// We need a MediaTracker to ensure our processing results in
// images that are ready for display
//MediaTracker mt = new MediaTracker(dummy);
if(theImage==null) // If we haven’t created the image, create it
{
timg = media.getImage(this,”graphics/”+imageName);
//mt.addImage(timg,1);
//mt.waitForID(1);
// We will use the dummy frame to create a duplicate of the
// image that we can do image processing on.
img =dummy.createImage(160,100);
Graphics g = img.getGraphics();
Color chromaKey = new Color(255,0,255);
g.setColor( chromaKey );
g.fillRect(0,0,160,100);
stampGraphic(g,160,100);
g.dispose();
theImage = ImageTools.mergeImages(timg,img,chromaKey);
}
return theImage;
}
catch(Exception e)
{ System.out.println(e); e.printStackTrace(System.out); return null; }
}
/**
* This method writes out specific card related data on top of the
* branding image, so the user can see the difference between two
* instances of the same type of Credit Card.
*
* @param img The image to stamp with specific information
*/
void stampGraphic(Graphics g, int w, int h)
{
try
{
Font f = new Font(“Helvetica”,Font.BOLD,11);
g.setColor(Color.black);
g.setFont(f);
FontMetrics fm = g.getFontMetrics();
Rectangle bounds = new Rectangle(0,0,w-1,h-1);
//String drawStr = localAlias+”\n\n\n”+PAN+”\n”+expireDate+
// “\n”+cardholderName;
```

```
String drawStr =
“\n\n”+PAN+”\n”+expireDate+”\n”+cardholderName;
TextDraw.drawCentered(g,fm,drawStr,bounds,2,0,TextDraw.CE
NTERED);
bounds = new Rectangle(1,1,w-1,h-1);
g.setColor( new Color(254,254,254) );
TextDraw.drawCentered(g,fm,drawStr,bounds,2,0,TextDraw.CE
NTERED);
}
catch(Exception e)
{ System.out.println(e); e.printStackTrace(System.out); }
}
/****************************************************************
* GenericCreditCard interface specific methods
***************************************************************/
/* Check if this instrument allows this purchase
* Has the option of returning false and keeping all data un-
available
* In this type of sceneraio, the data would be unavailable by
default,
* and would only become available if a valid accept() occurs...
* (e.g. — having a boolean OKAYTOGIVEOUTINFO; variable
that is checked
* by each getter().. set OKAYTOGIVEOUTINFO=true; only if
accept
* succeeds).
*
* @return true=is acceptable, false=is not acceptable
*/
public boolean accept()
{
// Has the option of returning false and keeping all data
unavailable
// In this type of scenario, the data would be unavailable by
default,
// and would only become available if a valid accept() occurs...
// (e.g. -- having a boolean OKAYTOGIVEOUTINFO; variable
that is checked
// by each getter().. set OKAYTOGIVEOUTINFO=true; only if
accept
// succeeds).
return true;
}
/**
* Returns the credit-card number
* @return the Instrument’s card number
*/
public String getPAN()
{ return PAN; }
/**
* Returns the card’s expiration date
* @return the Instrument’s expire date
*/
public String getExpireDate()
{ return expireDate; }
/**
* Returns the cardholder’s name as known by the issuing
institution
* @return the Instrument’s cardholder name
*/
public String getCardholderName()
{ return cardholderName; }
/**
* Return’s the cardholder’s billing address
* @return the Instrument’s cardholder address
*/
public AddressRecord getBillingAddress()
{ return billingAddress; }
/**
Creating Cassettes
* Sets the credit-card number for this instance
* @param the card number in string format
*/
public void setPAN(String x)
{ PAN = x; }
/**
* sets the card’s expiration date
*@param the expiration date as a string
*/
public void setExpireDate(String x)
{ expireDate = x; }
/** Sets the cardholder’s name as known by the issuing
institution
* the cardholder’s name
*/
public void setCardholderName (String x)
{ cardholderName = x; }
/**
* Sets the billing address as an AddressRecord
* @param The billing address as an AddressRecord
*/
public void setBillingAddress (AddressRecord x)
{ billingAddress = x; }
} // end of CCInstrument
```

The Protocol Cassette:

A protocol cassette handles the communications between a client and a server. An example of a protocol cassette is one that handles the SET (Secure Electronic Transaction) protocol. Because protocols vary so much, there is very little in common between different protocol cassettes. The Protocol interface defines the few methods that all protocol cassettes must share. The canUseInstrument method returns true if the protocol supports a particular instrument:

public boolean canUseInstrument(Instrument instrument)Sun recommends the following code structure when determining whether an instrument is supported:

```
if ( !(instrument instanceof NeededInterface_1) )
return false;
if ( !(instrument instanceof NeededInterface_2) )
return false;
if ( !(instrument instanceof NeededInterface_3) )
return false;
return true;
```

The getName method returns the name of the protocol as it was registered by the control cassette:

`public String getName()`

The setProtocolJCM tells the protocol cassette to read a JCM and extract information:

`public void setProtocolJCM(JCM protocolJCM)`

The setWalletGate method gives the protocol a WalletGate object, which is used to get permissionto perform particular operations:

`public void setWalletGate(WalletGate gate)`

According to Sun, the wallet gate and the protocol portion of the JCM may soon be merged into the CommerceContext object, so the setProtocolJCM and setWalletGate methods may disappear in future releases of the JCC API.The setCommerceContext method assigns a commerce context to the current protocol:public void setCommerceContext(CommerceContext context)

A commerce context contains information specific to the current operation. Sun predicts that future versions of JCC will assign the commerce context when the protocol is created, eliminating the need for the setCommerceContext method.

The PurchaseProtocol interface defines a typical protocol for making purchases. As complex as the purchase process is in an electronic environment, the PurchaseProtocol interface contains only one method:

```
        public boolean actUpon(Instrument instrument,
PurchaseParams purchase) throws TransactionException
```

Listing 13.2 shows an example protocol from Sun that is provided in the JCC package.

Listing 13.2 Source Code for DemoProtocol.java

```
/*
* @(#) @(#)DemoProtocol.java 1.7 11/07/97
*
* Copyright (c) 1996 Sun Microsystems, Inc. All Rights
Reserved.
*
* Permission to use, copy, modify, and distribute this software
* and its documentation for NON-COMMERCIAL or
COMMERCIAL purposes and
* without fee is hereby granted.
* Please refer to the file
http://java.sun.com/copy_trademarks.html
* for further important copyright and trademark information and
to
* http://java.sun.com/licensing.html for further important
licensing
* information for the Java (tm) Technology.
*


* SUN MAKES NO REPRESENTATIONS OR WARRANTIES
ABOUT THE SUITABILITY OF THE SOFTWARE, EITHER
EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO
THE IMPLIED WARRANTIES OF MERCHANTABILITY,
FITNESS FOR A PARTICULAR PURPOSE, OR NON-
INFRINGEMENT. SUN SHALL NOT BE LIABLE FOR ANY
DAMAGES SUFFERED BY LICENSEE AS A RESULT OF
USING, MODIFYING OR  DISTRIBUTING THIS SOFTWARE
OR ITS DERIVATIVES.

THIS SOFTWARE IS NOT DESIGNED OR INTENDED FOR
USE OR RESALE AS ON-LINE Creating Cassettes CONTROL
EQUIPMENT IN HAZARDOUS ENVIRONMENTS REQUIRING
FAIL-SAFE PERFORMANCE, SUCH AS IN THE OPERATION
OF NUCLEAR FACILITIES, AIRCRAFT NAVIGATION OR
COMMUNICATION SYSTEMS, AIR TRAFFIC CONTROL,
DIRECT LIFE
```

SUPPORT MACHINES, OR WEAPONS SYSTEMS, IN WHICH THE FAILURE OF THE SOFTWARE COULD LEAD DIRECTLY TO DEATH, PERSONAL INJURY, OR SEVERE PHYSICAL OR ENVIRONMENTAL DAMAGE (“HIGH RISK ACTIVITIES”). SUN SPECIFICALLY DISCLAIMS ANY EXPRESS OR IMPLIED WARRANTY OF FITNESS FOR HIGH RISK ACTIVITIES.

```
*/
// Things to be done to this code are marked with ?? or UNDONE:
package com.sun.commerce.example.demoprot;
import java.awt.*;
import java.awt.Event;
import java.io.*;
import java.net.*;
import java.security.*;
import java.util.*;
import javax.commerce.base.*;
import javax.commerce.base.WalletUserPermit;
import javax.commerce.cassette.*;
import javax.commerce.database.*;
import com.sun.commerce.gencc.GenericCreditCard;
import javax.commerce.gui.ProgressBar;
import javax.commerce.util.Money;
/**
* The DemoProtocol is a protocol that is intended to be used for
* Demo purposes. It does not actually perform a transaction or
* open any network connections. It mostly puts up a progress
* bar and pretends that it performed a transaction. This protocol
* will accept any instrument.
*
* @see PurchaseProtocol
* @see ActionBase
* @see ActionParameter
*/
public class DemoProtocol implements PurchaseProtocol, Runnable
{
/** The registration name of the protocol */
public static final String PROTOCOL_NAME=”Demo”;
/** The instrument being used in the demo protocol */
Instrument instrument;
private PurchaseParams pp;
private ProgressBar pbar;
private JCM jcm=null;
private String failure=null;
private boolean protocolDone = false;
private CommerceContext media;
/** Constructor
*/
public DemoProtocol() {}
/**
* Returns the registration name of the Demo Protocol.
*
* @return String The registration name of the Demo Protocol.
*/
public String getName() { return PROTOCOL_NAME; }
/**
* Called by the Operation to set the protocol’s JCM.
*
* @param prjcm The part of the JCM that applies to the
protocol.
*/
public void setProtocolJCM(JCM prjcm) { jcm=prjcm; }
public void setWalletGate(WalletGate wg)
{}
public void setCommerceContext(CommerceContext ccntxt)
{
media=ccntxt;
}
/**
* Pretends to perform the transaction. Uses the passed
instrument and
* parameters present a convincing progress bar.
*
* @param inst The CreditCard instrument used for the
transaction
*
* @param ap The parameters used for the payment
*
* @return boolean true= success
*/
synchronized public boolean actUpon(Instrument inst,
PurchaseParams pp)
throws TransactionException
{
// record parameters and double-check the instrument is of
// the correct type.
this.pp=pp;
pbar = new ProgressBar (media,"Paying "+
((javax.commerce.util.Invoice)pp.getInvoice()).getTotal().toString()+
" using "+pp.getInstrument().instrument.getDescription ());
pbar.setTitle (pp.getMerchant().getGeneralURL().toString ());
instrument=inst;
Creating Cassettes
if (true) {
System.out.println("Running Protocol in the same thread.");
commit();
} else {
System.out.println("Running Protocol as a sepparate thread.");
// Start the thread that will monitor the transaction
Thread t = JECF.makeThread(this);
t.start();
while (!protocolDone) {
try { wait();
} catch (InterruptedException ignored) {}
}
}
if (failure != null){
System.out.println("Failure: " + failure);
throw new TransactionException(failure);
}
// Return DONE -> An exception will be raised for an error condition
return true;
}
/**
* Check to see if this Protocol can use a specific instrument
*
* @param instrument The instrument to check
*
* @return boolean true=can use, false=cannot use
*/
public boolean canUseInstrument(Instrument instrument)
{
return true;
}
private void pause(int milliseconds) {
Thread thisThread = Thread.currentThread();
try { thisThread.sleep(500);
} catch (InterruptedException ignored) {}
}
```

```
private void setFailure(String newFailure) {
failure=newFailure;
if (newFailure != null) {
System.out.println(“setFailure(“ + failure + “)”);
throw new RuntimeException(failure);
}
}
private void checkCancelled (boolean cancelled) {
if (cancelled) {
setFailure(“User Cancelled Transaction”);
}
}
private void getJcmInfo() {
if (jcm!=null)
try { setFailure(jcm.getString(“failure”));
} catch (JCMException ignored) { }
if (jcm!=null) {
System.out.println(“Displaying JCM:”);
Enumeration leaves = jcm.elements() ;
while (leaves.hasMoreElements()) {
String[] leaf = (String[]) leaves.nextElement();
if (leaf != null) {
System.out.print(“ “ + leaf[0] + “=”);
for (int i=1; i<leaf.length; i++) {
System.out.print(leaf[i] + “ “);
}
System.out.println(“”);
}
}
} else {
System.out.println(“No JCM was set!”);
}
}
private Random rand = new Random();
int rand(int minValue, int maxValue) {
return minValue + Math.abs(rand.nextInt()%(maxValue-
minValue));
}
synchronized protected boolean commit()
{
try {
getJcmInfo();
// Start the progress bar
pause(500);
checkCancelled(pbar.showProgress (0, “Initiating Communications”));
for (int i=rand(5,15); i<100; i+=rand(5,15)) {
pause(rand(10,500));
checkCancelled(pbar.showProgress (i, “Transaction is “ + i + “% done.”));
}
pause(500);
pbar.showProgress (100, “Transaction is complete.”);
return true;
} catch (RuntimeException ignored) {
return false;
} finally {
protocolDone=true;
notify();
pbar.done ();
}
}
public void run(){
commit();
}
}
```

The Operation Cassette :

The operation cassette is really the hub of JCC. When an operation is selected, the operation’s user interface collects the information needed to complete the transaction, gets the necessary instruments, and activates a protocol to complete the deal. As with the rest of the Java Commerce Client, the information in this section is subject to change. Sun suggests that some of the information in the Operation interface will be merged into the CommerceContext object in a future release. The Operation interface and its partners represent a pretty complex system that takes sometime to adjust to. Most of the methods in the Operation interface itself are used to set up the initial context for the operation. The setJCM gives the Operation object a JCM to parse and retrieve information related to the operation:

```
public void setJCM(JCM message) throws Exception
```

The getJCMDescription method returns a description of the required and optional parameters for this operation:

```
public String[] getJCMDescription()
```

The description strings are returned in the form “name=description”.

The setWalletGate method gives the operation a WalletGate, which it uses to get varioussecurity permissions:

```
public void setWalletGate(WalletGate gate)
```

The setID method gives the operation its row ID from the database:

```
public void setID(RowID id)
```

The setCommerceContext method sets the current context for the operation:

```
public void setCommerceContext(CommerceContext context)
```

In the future, some of the other parameters may be merged into the commerce context, andthe context itself may be part of the Constructor for the operation.The execute method starts the operation, which then brings up a UI object to collect the informationfor the operation:

```
public String execute() throws Exception
```

The Service Cassette:

A service cassette is a utility cassette used by other cassettes within JCC. There is no service interface, but there is a ServiceUI because most service cassettes perform user interface functions. In addition, the ServiceUI interface is used by the Operation interface for a user interface.

The getClientContainer method in the ServiceUI interface returns the container that represents the user interface:

```
public Container getClientContainer(CommerceContext context, Dimension dim)
```

The getSelectedImage and getUnselectedImage methods return the images used to represent the service if it is shown inside another container as something to be selected:

```
public Image getSelectedImage(CommerceContext context, Dimension dim)
```

```
public Image getUnselectedImage(CommerceContext context, Dimension dim)
```

The getSelectorText returns the text that serves as a label for the image when shown as animage:

```
public String getSelectorText()
```

Finally, the setWalletGate method gives the ServiceUI a WalletGate object for security perations:

```
public void setWalletGate(WalletGate gate)
```

Listing 13.3 shows an example Rolodex service from Sun’s examples in the JCC package.Listing 13.3 Source Code for Rolodex.java

```
/*
* @(#)Rolodex.java 1.1 97/10/29
*
* Copyright (c) 1996 Sun Microsystems, Inc. All Rights Reserved.
```

```
*
* Permission to use, copy, modify, and distribute this software
* and its documentation for NON-COMMERCIAL or
COMMERCIAL purposes and
* without fee is hereby granted.
* Please refer to the file
http://java.sun.com/copy_trademarks.html
* for further important copyright and trademark information and
to
* http://java.sun.com/licensing.html for further important
licensing
* information for the Java (tm) Technology.
*
* SUN MAKES NO REPRESENTATIONS OR WARRANTIES
ABOUT THE SUITABILITY OF THE SOFTWARE, EITHER
EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO
THE IMPLIED WARRANTIES OF MERCHANTABILITY,
FITNESS FOR A PARTICULAR PURPOSE, OR NON-
INFRINGEMENT. SUN SHALL NOT BE LIABLE FOR ANY
DAMAGES SUFFERED BY LICENSEE AS A RESULT OF
USING, MODIFYING OR  DISTRIBUTING THIS SOFTWARE
OR ITS DERIVATIVES.
THIS SOFTWARE IS NOT DESIGNED OR INTENDED FOR
USE OR RESALE AS ON-LINE CONTROL EQUIPMENT IN
HAZARDOUS ENVIRONMENTS REQUIRING FAIL-SAFE
PERFORMANCE, SUCH AS IN THE OPERATION OF
NUCLEAR FACILITIES, AIRCRAFT NAVIGATION OR
COMMUNICATION SYSTEMS, AIR TRAFFIC CONTROL,
DIRECT LIFE SUPPORT MACHINES, OR WEAPONS
SYSTEMS, IN WHICH THE FAILURE OF THE SOFTWARE
COULD LEAD DIRECTLY TO DEATH, PERSONAL INJURY,
OR SEVERE PHYSICAL OR ENVIRONMENTAL DAMAGE
(“HIGH RISK ACTIVITIES”). SUN  SPECIFICALLY DISCLAIMS
ANY EXPRESS OR IMPLIED WARRANTY OF FITNESS FOR
HIGH RISK ACTIVITIES.

Creating Cassettes
*/
package com.sun.commerce.example.rolodex ;
import java.awt.*;
import javax.commerce.database.* ;
import javax.commerce.cassette.* ;
import javax.commerce.util.*;
import javax.commerce.base.*;
import java.util.* ;
import java.io.* ;
import java.security.* ;
import javax.commerce.base.Constants;
import javax.commerce.gui.InfoDialog;
/**
* @author Surya Koneru
* @(#)Rolodex.java 1.1 97/10/29
*/
public class Rolodex implements ServiceUI
{
public static final String SERVICE_NAME = new
String(“Rolodex”);
public WalletGate wgate;
private WalletAdminPermit wap;
private DatabaseOwnerPermit dop;
/**
* This method will always be called before any other ServiceUI
methods
*
* @param gate A WalletGate to allow the service to utilize
wallet-level
* functionality.
*/
public void setWalletGate(WalletGate gate)
{
wgate=gate;
wap = gate.getWalletAdminPermit( new Ticket( “W_OWNER”));
dop = wap.getDatabaseOwnerPermit();
}
/**
* This method fetches a service’s client container for display
within
* the encompassing WalletUI. The ServiceUI author is urged to
use
* commerce widgets, but it is not neccessary.<br><br>
*
* This method, in general, returns an a light-weight container
that
* has transparent characteristics. It may,
* however, return any Container.
*
* @param ccontext A CommerceContext that may be used to
fetch imagery
* and for use in constructing the ServiceUI. The
* UIFactory of this CommerceContext models the
* current WalletUI.
* @param hint The initial dimensions of the Service UI. The
returned
```

```
* container is likely to be resized many times, however.
*
* @return Container The AWT Container that represents the
Service’s UI.
* This Container should have dimensions that match
* the passed in hint.
*/
public Container getClientContainer(CommerceContext
ccontext,
Dimension hint)
{
return new RolodexPanel(ccontext,dop);
}
/**
* Used to fetch the image displayed on the WalletUI selector
when the
* selector for this service is not selected.
*
* @param factory A CommerceContext that may be used to
fetch imagery.
* @param size The size of the requested image (Generally
16x16 pixels)
*
* @return Image The image to display when this serivce is not
selected.
*/
public Image getUnselectedImage(CommerceContext factory,
Dimension size)
{ return null; }
/**
* Used to fetch the image displayed on the WalletUI selector
when the
* selector for this service is selected.
*
* @param factory A CommerceContext that may be used to
fetch imagery.
* @param size The size of the requested image (Generally
16x16 pixels)
*
* @return Image — The image to display when this service is
selected.
*/
public Image getSelectedImage(CommerceContext factory,
Dimension size)
{ return null; }
```

```
/**
* Used to fetch the text that will be displayed on the WalletUI selector.
*
* @return String -- The text to display on the WalletUI selector for this
* service.
*/
public String getSelectorText() { return SERVICE_NAME; }
}
```

The User Interface Cassette:

A user interface cassette contains a WalletUI object, a TransactionListener, anActionListener, and a UIFactory. The WalletUI interface defines a set of methods that indicate which operations and servicesare available in the UI, and also information about the current security context. TheaddOperation method adds an operation to the operations available from this user interfaceand returns a unique index number:

```
public int addOperation(Operation op)
```

The canUseOperation method returns true if the user interface cassette is compatible with aparticular operation:

```
public boolean canUseOperation(Operation op)
```

The addSelector method adds a service to the available services and returns a unique indexnumber:

```
public int addSelector(ServiceUI service)
```

The removeSelector method removes a service from the user interface:

```
public void removeSelector(int index)
```

The addSelector and removeSelector methods deal with objects visible from the user interface.If an operation is added that has a user interface component, the addOperation methodwill likely call the addSelector method to add it to the visible items. Thus if an operation has avisible component, you can use removeSelector to remove it because there is noremoveOperation method.The setCommerceContext method sets the current context for the user interface and should becalled immediately after the user interface is instantiated:

```
public void setCommerceContect(CommerceContext context)
```

]The init method must be called after the commerce context has been set, but before anyother operations are performed:public void init()

The getName method returns the name of the user interface cassette as it was registered by theCassetteControl class:

public String getName()

The select method selects a different object within the user interface:

public void select(int selectorIndex)

The selection value should be the unique index of one of the available selectors.The populate method tells the user interface to draw itself in an AWT container:

public void populate(Container cont)

Bear in mind that the populate method might be operating on a visible container. Therefore, if you do ugly things such as adding a bunch of objects and then removing them, your user interfaceis liable to appear a little wacky.

Listing 13.4 shows a demo user interface cassette from Sun’s JCC examples.

Listing 13.4 Source Code for DemoUI.java

```
/*
* @(#)DemoUI.java 1.23 11/07/97
*
* Copyright (c) 1997 Sun Microsystems, Inc. All Rights Reserved.
*
* Permission to use, copy, modify, and distribute this software
* and its documentation for NON-COMMERCIAL or COMMERCIAL purposes and
* without fee is hereby granted.
* Please refer to the file http://java.sun.com/copy_trademarks.html
* for further important copyright and trademark information and to
* http://java.sun.com/licensing.html for further important licensing
* information for the Java (tm) Technology.
*
* SUN MAKES NO REPRESENTATIONS OR WARRANTIES ABOUT THE SUITABILITY OF * THE SOFTWARE, EITHER EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE IMPLIED WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, OR NON-
```

```
INFRINGEMENT. SUN SHALL NOT BE LIABLE FOR ANY
DAMAGES SUFFERED BY LICENSEE AS A RESULT OF
USING, MODIFYING OR  DISTRIBUTING THIS SOFTWARE
OR ITS DERIVATIVES. THIS SOFTWARE IS NOT DESIGNED
OR INTENDED FOR USE OR RESALE AS ON-LINE
CONTROL EQUIPMENT IN HAZARDOUS ENVIRONMENTS
REQUIRING FAIL-SAFE PERFORMANCE, SUCH AS IN THE
OPERATION OF NUCLEAR FACILITIES, AIRCRAFT
NAVIGATION OR COMMUNICATION SYSTEMS, AIR
TRAFFIC CONTROL, DIRECT LIFE SUPPORT MACHINES,
OR WEAPONS SYSTEMS, IN WHICH THE FAILURE OF THE
SOFTWARE COULD LEAD DIRECTLY TO DEATH,
PERSONAL INJURY, OR SEVERE PHYSICAL OR
ENVIRONMENTAL DAMAGE (“HIGH RISK ACTIVITIES”). SUN
SPECIFICALLY DISCLAIMS ANY EXPRESS OR IMPLIED
WARRANTY OF FITNESS FOR HIGH RISK ACTIVITIES.
*/
package com.sun.commerce.example.demoui ;
import javax.commerce.database.* ;
import javax.commerce.cassette.* ;
import javax.commerce.util.*;
import java.util.* ;
import java.io.* ;
import java.security.* ;
import javax.commerce.base.*;
import javax.commerce.gui.*;
import java.awt.*;
import java.awt.event.*;
/**
* @author Daniel J. Guinan
* @version @(#)DemoUI.java 1.23 11/07/97
*
*/
public class DemoUI implements WalletUI, ActionListener,
TransactionListener
{
class ServiceNode
{
Creating Cassettes
public ServiceUI service;
public Container container;
public CWSelector button;
public int selectorIndex;
public ServiceNode(ServiceUI sui, CommerceContext media,
String buttontext, int index)
{
```

```
service=sui;
container=null;
selectorIndex=index;
Dimension d = new Dimension(20,20);
button=new CWSelector(media,buttontext,
sui.getUnselectedImage(media,d),
sui.getSelectedImage(media,d));
}
}
public static final String WALLETUI_NAME = “DemoUI”;
public static final String WALLETUI_DESCRIPT = “Demonstration
 Wallet User Interface”;
public static final Color INACTIVE_COLOR = Color.lightGray;
public static final Color ACTIVE_COLOR = new Color(150,175,255);
private CommerceContext context;
private UIFactory widgets;
private CWPanel loadingPanel;
private CWPanel selectionPanel;
private Container currentClient = null;
private CWPanel masterContainer = null;
// private CWSelector selectedButton = null;
private ServiceNode currentlySelected = null;
private Hashtable serviceTable = new Hashtable(); // (name, ServiceNode)
private Hashtable idxMapping = new Hashtable(); // (Integer, name)
private Hashtable pendingTable = new Hashtable(); // (Operation, name)
private int currentIdx = 0;
////////////////////////////////////////////////////////////
///////////////// WalletUI Interface Methods /////////////////
////////////////////////////////////////////////////////////
/**
* Init() is called after the current CommerceContext and
* CommerceUIFactory objects are set, but before any other
* calls are made into the WalletUI. It is here that initialization
* code should be executed.<br><br>
*
* PLEASE NOTE: This is not where the UI is shown. That occurs
* in the populate() method.
*/
```

```
public void init()
{
}
/**
* Retrieves the current CommerceContext.
*
* @return CommerceContext -- The current CommerceContext
*/
public CommerceContext getCommerceContext()
{ return context; }
/**
* Sets the current CommerceContext (THIS METHOD IS
ALWAYS
* CALLED DIRECTLY AFTER OBJECT INSTANTIATION)
*
* @param ccontext The CommerceContext to use as current.
*/
public void setCommerceContext(CommerceContext ccontext)
{
context=ccontext;
new CommonGraphics(ccontext); // Load static common
graphics...
widgets=new DemoUIFactory();
context.setUIFactory( widgets );
loadingPanel = new CWPanel(ccontext);
loadingPanel.setLayout(new BorderLayout());
currentClient=loadingPanel;
CWLabel loading = new
CWLabel(ccontext,”Loading...”,CWLabel.CENTER);
loading.setFont(new Font(“GilSans”,Font.BOLD,20));
loading.setForeground(Color.white);
loadingPanel.add(loading,BorderLayout.CENTER);
}
/**
* This method is called to add a Service selector.
*
* @param service The ServiceUI being added.
* @return int -- The unique index of the selector created.
*/
public int addSelector(ServiceUI service)
{
System.out.println(“DEMOUI: Adding service =
“+service.getSelectorText());
int ret=currentIdx++;
```

```
String selectorText = service.getSelectorText();
//UNDONE: Freak out if the selectorText is already there..
if( serviceTable.get(selectorText)!=null )
{
boolean unique=false;
int index=1;
String name=null;
while(!unique)
Creating Cassettes
{
name = selectorText+” “+(++index);
if(serviceTable.get(name)==null) unique=true;
}
selectorText=name;
}
//return -1;
idxMapping.put( new Integer(ret), selectorText );
ServiceNode sn = new ServiceNode(service,context,selectorText,ret);
serviceTable.put(selectorText, sn);
selectionPanel.add(sn.button);
sn.button.addActionListener(this);
// Do we really want to do this??
selectionPanel.validateAll();
return ret;
}
/**
*
* @param op The Operation that should have a selector added for it.
* @return int — The unique index of the selector created.
*/
public int addOperation(Operation op)
{
int idx = -1;
if(op instanceof ServiceUI)
{
idx= addSelector((ServiceUI)op);
select(idx);
String name = (String)idxMapping.get( new Integer(idx) );
ServiceNode sn = (ServiceNode)serviceTable.get(name);
sn.button.setPending(true);
pendingTable.put(op,name);
op.addTransactionListener(this);
```

```
}
//UNDONE: Exception??
return idx;
}
public void transactionPerformed(TransactionEvent evt)
{
Operation source = (Operation)evt.getSource();
String name = (String)pendingTable.get(source);
if(name==null) return;
pendingTable.remove(source);
ServiceNode sn = (ServiceNode)serviceTable.get(name);
sn.button.setPending(false);
}
/**
* This method is called to remove a Service selector.
*
* @param int The unique index of the Service selector to
remove. This
* is the same index returned by addSelector().
*/
public void removeSelector(int idx)
{
Integer i = new Integer(idx);
String stext = (String)idxMapping.get(i);
if(stext==null)
{
System.out.println("Cannot remove selector: Index "+idx+
" is not valid!");
return; //Not there??
}
ServiceNode sn = (ServiceNode)serviceTable.get(stext);
System.out.println(">> Removing Service "+stext);
// selectionPanel.setCanInvalidate(false);
selectionPanel.remove(sn.button);
sn.button.removeActionListener(this);
idxMapping.remove(i);
serviceTable.remove(stext);
selectionPanel.validateAll(); // we removed a button...
if(sn.selectorIndex==idx)
{
//If this guy is currently selected, change...
Enumeration e = serviceTable.elements();
sn = (ServiceNode)e.nextElement();
if(sn!=null) select(sn.selectorIndex);
```

```
else showClient(null);
}
}
/**
* This method is called to force the WalletUI to change it’s focus to
* a particular service.
*
* @param idx The uniqe index of the Service selector to select. This
* is the same index returned by addSelector().
*/
public void select(int idx)
{
Integer i = new Integer(idx);
String stext = (String)idxMapping.get(i);
if(stext==null) return; // Not there??
showClient(stext);
}
```

Creating Cassettes

```
/**
* This method is used to determine compatibility with Operations. When
* Operations or WalletUIs are installed, this method is called on all
* WalletUIs against all Operations to generate a compatibility list.
* <br><br>
*
* It is this compatibility list that appears in the user’s preferences,
* allowing a user to change their “preferred” UI for various operations.
* <br><br>
*
* In general, this method will check interfaces of the Operation and return
* true if it feels that it can accomidate the Operation’s UI requirements.
* <br><br>
*
* @param op The operation to check for UI compatibility.
* @return boolean — true=compatible with operation, false=not compatible
*
```

```
* @see WalletUI.addOperation
*/
public boolean canUseOperation(Operation op)
{
//The demonstration UI works with any operation that implements ServiceUI...
if(op instanceof ServiceUI) return true;
else return false;
}
/**
* Retrieves the registration name of this WalletUI. This name must match
* the registration name used in the CassetteControl.install() method when
* registering this UI.
*
* @return String -- The registration name of this WalletUI.
*/
public String getName()
{
return WALLETUI_NAME;
}
/**
* This method is called to present the WalletUI. In general, this happens
* right before the JECF performs a show() on the frame that contains the
* various widgets that represent the WalletUI. The whole of the WalletUI
* will be shown within the container passed in this method. The following
* considerations should be carefully taken into account by WalletUI cassette
* writers:<br><br>
*
* <li> The container passed to this method is probably, but not
* necessarily a Frame or an Object that inherets Frame.
* (NOTE: Never cast this Container to Frame)
* <li> The container may change dimensions. It is expected that the
* User will resize the Window, causing this container to resize
* as a result. The UI must be written to accommodate this (i.e.
* stretch).
*
```

```
* <li> Do not rely on any characteristics of any AWT components
that
* may exist above this Container. Those characteristics are
* subject to change at any time, forcing any cassettes that rely
* upon them to become obsolete.
*
* <li> The populate method may be populating a live AWT
component
* (i.e. already shown).
* </ul>
*
* @param c The container that the WalletUI should be drawn
within.
*/
public void populate(Container c)
{
masterContainer = new CWPanel(context);
c.setLayout(new BorderLayout(0,0));
c.add(masterContainer,BorderLayout.CENTER);
masterContainer.setLayout(new BorderLayout(0,0));
currentClient=loadingPanel;
masterContainer.add(currentClient,BorderLayout.CENTER);
loadingPanel.validateAll();
selectionPanel = new SelectionPanel(context);
selectionPanel.setLayout( new VFlowLayout(VFlowLayout.
 TOP|VFlowLayout.HORZ_LEFT) );
masterContainer.add(selectionPanel,BorderLayout.EAST);
}
/////////////////////////////////////////////////////////////
////////////////////// Private Methods //////////////////////
/////////////////////////////////////////////////////////////
public void actionPerformed(ActionEvent evt)
{
Object source = evt.getSource();
if(source instanceof CWSelector)
{
CWSelector bt = (CWSelector)source;
String item = bt.getLabel();
showClient(item);
//cardLayout.show(switchPanel,item);
}
}
private void showClient(String stext)
{
Container newClient;
```

```
// Tell the selector it is not current anymore.
if(currentlySelected!=null)
currentlySelected.button.setCurrent(false);
if(stext==null) // What is this??
{
newClient = loadingPanel;
currentlySelected=null;
}
else // Get the new client
{
ServiceNode sn = (ServiceNode)serviceTable.get(stext);
newClient = sn.container;
currentlySelected = sn;
// Tell the new selector it is not current anymore.
sn.button.setCurrent(true);
}
// If this is the first time, do something special.
if(newClient == null) firstTimeShowClient(stext);
else // Just change the page
{
//if(currentClient!=null)
masterContainer.swapComponentValid(currentClient,newClient,
BorderLayout.CENTER);
currentClient=newClient;
}
//reLayoutPaintMaster();
}
/* void reLayoutPaintMaster()
{
// Relayout and repaint everything...
if(masterContainer instanceof CWBasePanel)
((CWBasePanel)masterContainer).validateAll();
else if(masterContainer instanceof CWidget)
((CWidget)masterContainer).validateAll();
else { masterContainer.validate(); masterContainer.repaint(); }
}*/
private void firstTimeShowClient(String stext)
{
// Show the loading loadingPanel
if(currentClient!=loadingPanel)
{
masterContainer.swapComponentValid(currentClient,loadingPa
nel,
BorderLayout.CENTER);
currentClient=loadingPanel;
context.showStatus(“Loading...”);
}
// Do the normal stuff...
ServiceNode sn = (ServiceNode)serviceTable.get(stext);
Container clientArea =
sn.service.getClientContainer(context,
loadingPanel.getSize() );
clientArea.setSize(loadingPanel.getSize());
clientArea.setLocation(loadingPanel.getLocation());
sn.container=clientArea;
// swap
masterContainer.swapComponentValid(currentClient,clientArea,
BorderLayout.CENTER);
currentClient=clientArea;
context.showStatus("");
}
}
```

The Java Commerce Client is a very new API and will likely go through several changes before it really makes an impact on the Internet. Although this chapter has covered the construction of commerce client objects, you will probably not have to write your own commerce objects. Instead, you should be able to get cassettes from various vendors and online stores. If you are creating your own Shopping Cart applet, however, you will probably want to write your own UI cassette and still leave the lower-level cassettes to other developers.

13.12 SUMMARY

This chapter coversWhat Is the Java Media Framework, Creating a Media Player, adding the Player to Your Application, Compiling the Basic Player, The States of the Player, Adding Controls to the Player, Controlling the Player Programmatically, Linking Multiple Players, Creating Your Own Media Stream, Commerce and Java Wallet, Security Support with the JCC, Commerce Messages, Creating Cassettes.

13.13 QUESTIONS

1. What Is the Java Media Framework?
2. How will you create a Media Player?
3. What are the States of the Player?
4. How will you add Controls to the Player?
5. How will you control the Player Programmatically?
6. How will you link Multiple Players?
7. How will creating Your Own Media Stream?
8. Write a note on "Java Commerce Messages.
9. How will you create Cassettes?

*****

14

DATA STRUCTURES AND JAVA UTILITIES

Unit Structure

14.1 WHAT ARE DATA STRUCTURES?

The java.util package provides several useful classes that give important functionality to the Java runtime environment. These classes provide much of the code that you frequently end up writing yourself when you write in C++. The creators of Java realized that one of the things people really like about Smalltalk is the abundance of useful utility classes. The java.util package focuses mostly on container objects—that is, objects that contain or hold other objects. In addition to the containers, the package also adds a handy utility class for breaking up a string into words (tokens), expanded support for random numbers and dates, and a carryover from Smalltalk called *observables*.

Data structure is a general computer science term for an object that holds a collection of other objects. For instance, an array is the simplest data structure. Data structures can vary in complexity, but ultimately their goal is to hold and manipulate objects. Some data structures, like arrays, keep data in one long list. Others, like trees, keep the data sorted in non-linear storage compartments. Each type of data structure has its advantages and disadvantages. For instance, trees are extremely efficient at finding and inserting sorted data, while hash tables are even more efficient at finding data, but at the cost of more memory usage.

14.2 COLLECTIONS

One significant design change in the Java 1.2 API is the creation of a group of classes called the Collection API. The Collection API provides a common set of interfaces for all data structures in the java.util package. Before collections, converting between one data structure and another required some nontrivial amount of work. However, the Collection API provides a uniform mechanism for doing this. In addition, the Collection API is designed to allow characteristics, such as the ordering of an object, to be used in many types of structures. Now you can choose the proper data structure based on its performance characteristics, without having to worry about the implementation mechanics.

Collection Interface:

At the root of the Collection API is the Collection interface. The goal of the Collection interface is to provide all the common methods all collection classes will have. Now, not 100 percent of all collections will actually provide an implementation for all of the methods. If a method isn’t implemented in a Collection and you try to call that method anyway, the method will probably throw an UnsupportedOperationException.As you might expect, the Collection interface provides mechanisms for inserting new objects into the collection. The first method allows you to insert a single element. The second allows you to add all of the elements in another collection.

boolean add(Object o)

boolean addAll(Collection c)

To get objects out of the collection, you can choose from three methods. The first two return the contents of the container in an array. The interesting thing about the second of these methods is that you pass into it an array of the object type you wish to return. The last of these methods returns a new class called Iterator. You’ll see how Iterator works later in this chapter.

Object[] toArray()
Object[] toArray(Object[] a)
Iterator iterator()

You can also remove objects from the collection, either by emptying the array (clear()) or by removing a specific object (remove()). In addition, you can remove a set of objects by removing all the objects in the collection that also exist in the collection passed as a parameter; or you can retain the objects in the collection that’s passed in and remove all the rest of the elements.

void clear()
boolean remove(Object o)
boolean removeAll(Collection c)
boolean retainAll(Collection c)

The final set of methods allows you to check the status of the collection. The first two allow you to determine if an object or a set of objects is included in a collection. The third method determines if the collection is empty, and the fourth returns the number of elements in the collection.

boolean contains(Object o)
boolean containsAll(Collection c)
boolean isEmpty()
int size()

The Collection interface is implemented by several classes throughout this chapter, so you'll find examples of how to use each of these methods later.

List Interface:

The Collection interface is extended and specialized with two sub-interfaces, List and Map. The List interface builds on Collection and adds some methods for a collection that store the objects in order. The creation of the java.util.List interface in the 1.2 API has created a name conflictwith java.awt.List. If you import both java.awt.* and java.util.*, you will have to do some extra work to use the List. You can do this two ways: by importing java.util.Listspecifically, or by using the fully qualified name of the List (java.util.List) instead of just List.

The primary enhancement that an ordered list adds is the concept of an index. The index is the location where the object is actually stored. So, the List adds several methods that deal with this concept.

The two new add() methods allow you to insert elements at a particular index and shift the elements after that index down so that the remaining objects are shifted in the collection to come after the new object(s). In addition, you can now substitute an object at a particular index with the new one, using the set() method.

void add(int index, Object element)
boolean addAll(int index, Collection c)
Object set(int index, Object element)

You can also get the objects out of the container in three new ways. The first two return a newiterator called a ListIterator, which provides bidirectional access. The first of these two methodsreturns a ListIterator of the entire collection, the second returns just an iterator of the collection starting at the specified index. The third method allows you to retrieve just the object at the specified index.

ListIterator listIterator()
ListIterator listIterator(int index)
Object get(int index)

Obviously you will also want to be able to remove objects from the collection in some new waystoo. The first of the new methods removes the element at the specified index, while the secondremoves all of the objects from the from Index up to (but not including) the to Index.

Object remove(int index)
void removeRange(int fromIndex, int toIndex)

Finally, it is often useful to know the index where an element is positioned in the collection.

The three methods for getting the element will return either the index where the element is stored, or, if the element is not found, they will return –1. The first of these methods returns the first index where a matching object is found and starts to look at the startingIndex. The second simply returns the last index where the object can be found, and the last method returns the last index, so long as it is not less than the minIndex.

int indexOf(Object o, int startingIndex)
int lastIndexOf(Object o)
int lastIndexOf(Object o, int minIndex)

Map Interface:

The second interface which extends from Collection is the Map interface. Unlike a List, a Map ensures that there will be at most one instance of an object and at most one null in the collection.

Map contains an element and a key value; the key determines where in the Map the element should be placed, and the keys can not be duplicated in the Map.

To insert elements into Map you must provide both the key and the value. So, obviously a new method is required to do just that (put()). In addition, because Map needs to know the key, it’s not possible to just copy any collection and insert it into the collection. Instead, the putAll() method copies the elements and uses their associated keys.

Object put(Object key, Object value)
void putAll(Map t)

To get objects out of a Map, you generally don’t need to know the object itself, but rather its key. The get() method returns the element with the matching key. In addition, Map can return a new interface called a Set. The two methods return either the Set for the entries or the keys.

The last new method for getting the elements in the collection returns a regular collection of the elements in Map.
Object get(Object key)

Set entrySet()
Set keySet()
Collection values()

Like inserting new elements into Map, removing elements is done via the key value, not the element itself via the remove() method.

Object remove(Object key)

The last of the new Map methods determines if a specified key or element is found. The containsKey() method returns true if the key exists in Map, while containsValue() returns true if the indicated object is mapped in Map via one or more keys.

boolean containsKey(Object key)

boolean containsValue(Object value)

Iterator Interface :

As you've seen, a Collection allows you to view its contents via a new interface called the Iterator interface. If you are familiar with the Enumeration interface, which has existed in the Java API since the 1.0 version, the Iterator will be very familiar. However, the Iterator also allows you to remove an element from the underlying collection.

The three methods first allow you to know if there are any additional elements in Iterator.

boolean hasNext()

Second, Iterator will return the next element in Iterator.

Object next()

Finally, you can remove the last element read from Iterator from the underlying collection.

void remove()

ListIterator Interface

A List has the ability to provide a more specific form of the Iterator. The ListIterator takesadvantage of the indexing in the List and allows you to perform several additional operations.

The first new capability is the ability to insert a new object, or to change the value of the oneyou just read. So add() will insert a new object, while set() replaces the object that was justread out of the Iterator with the one specified.

void add(Object o)

void set(Object o)

Collections

In addition to being able to read the next element in Iterator, a ListIterator also allows youto read the previous element. So ListIterator adds hasPrevious() in addition to thehasNext() method.boolean hasPrevious()

Obviously you will also want to retrieve the previous object as well, so, in addition to the next()method, a ListIterator also has a previous() method.

Object previous()

Finally, since you're looking at a list, you can also get the index values of the next or previouselements.

int nextIndex()
int previousIndex()

14.3 THE VECTOR CLASS

Java arrays are powerful, but they don't always fit your needs. Sometimes you want to put itemsin an array, but you don't know how many items you will be getting. One way to solve this is tocreate an array larger than you think you'll need. This was the typical approach in the days of Cprogramming. The Vector class gives you an alternative to this. A *vector* is similar to an arrayin that it holds multiple objects, and you retrieve the objects using an index value. The bigdifference between arrays and vectors is that vectors automatically grow when they run out ofroom. They also provide extra methods for adding and removing elements that you wouldnormally have to do manually in an array, such as inserting an element between two others.Effectively, a vector is an extensible array.

Before the 1.2 API, Vector extended just Object. Now with the 1.2 API, it extends fromAbstractList, which implements List.

Creating a Vector:

When you create a vector, you can specify how big it should be initially and how fast it shouldgrow. You can also just set the vector's initial size and let it figure out how fast to grow, or youcan let the vector decide everything for itself. To accomplish these various forms of initialization,the Vector class has three constructors.

public Vector()

creates an empty vector.

public Vector(int initialCapacity)

creates a vector with space for initialCapacity elements.

public Vector(int initialCapacity, int capacityIncrement)

creates a vector with space for initialCapacity elements. Whenever the vector needs to grow,

it grows by capacityIncrement elements.

JDK 1.2 adds one more constructor to support the Collection API, discussed at the end ofthis chapter. The new constructor creates a vector that initially has all the elements in theCollection, in the order they appear from the Collection's Iterator.

public Vector(Collection c)

If you have some idea of the typical number of elements you will be adding, go ahead and setup the vector with space for that many elements. If you don't use all the space, that's okay; youjust don't want the vector to have to allocate more space over and over.

Adding Objects to a Vector:

In addition to the standard List methods, there are two ways to add new objects to a vector.You can add an object as the last element in the vector, or you can insert an object between twoexisting objects. The addElement method adds an object as the last element:

Public final synchronized void addElement(Object newElement)

The insertElementAt() method adds a new object at a specific position. The index parameterindicates where in the vector the new object should be placed:

public final synchronized void insertElementAt(Object newElement, int index)throws ArrayIndexOutOfBoundsException

If you try to insert the new element at a position that does not exist yet—for example, if you tryto insert at position 9 and there are only five elements in the vector—you get anArrayIndexOutOfBoundsException.

You can change the object at a specific position in the vector with the setElementAt method:

public final synchronized void setElementAt(Object ob, int index)throws ArrayOutOfBoundsException

This method works almost exactly like the insertElementAt method, except that the otherelements in the vector are not shifted over to make room for a new object. In other words, thenew object replaces the old one in the vector.

Accessing Objects in a Vector:

Unfortunately, accessing objects in a vector is not as simple as accessing array elements. Insteadof giving an index surrounded by brackets ([]), you use the elementAt method to accessThe Vector Classvector elements. The vector equivalent of someArray[4] is someVector.elementAt(4). Theformat of the elementAt method is:

public final synchronized Object elementAt(int index)

throws ArrayIndexOutOfBoundsException

You can also access the first and last elements in a vector with the firstElement andlastElement methods:

public final synchronized Object firstElement()

throws NoSuchElementException

public final synchronized Object lastElement()

throws NoSuchElementException

If no objects are stored in the vector, these methods both throw a NoSuchElementException.

You can test to see whether a vector has no elements using the isEmpty method:

public final boolean isEmpty()

Many times you want to use a vector to build up a container of objects but then convert thevector over to a Java array for speed purposes. You usually only do this after you have all theobjects you need. For instance, if you are reading objects from a file that can contain any numberof objects, you store the objects in a vector. When you have finished reading the file, youcreate an array of objects and copy them out of the vector. The size method tells you howmany objects are stored in the vector:public final int size()

After you know the size of the vector, you can create an array of objects using this size. TheVector class provides a handy method for copying all the objects in a vector into an array ofobjects:

public final synchronized void copyInto(Object[] obArray)

If you try to copy more objects into the array than it can hold, you get anArrayIndexOutOfBounds exception. The following code fragment creates an object array andcopies the contents of a vector called myVector into it:

Object obArray[] = new Object[myVector.size()]; // Create object array

myVector.copyInto(obArray); // Copy the vector into the array

The Enumeration Interface

If you want to cycle through all the elements in a vector, you can use the elements method toget an Enumeration object for the vector. An Enumeration is responsible for accessing elementsin a data structure sequentially. It contains two methods.

public abstract boolean hasMoreElements()

returns true while there are still more elements to access. When there are no more elementsleft, this method returns false.

public abstract Object nextElement()

throws NoSuchElementException

returns a reference to the next element in the data structure. If there are no more elements toaccess and you call this method again, you get a NoSuchElementException.In the case of the Vector class, the elements() method returns an Enumeration interface forthe vector:

public final synchronized Enumeration elements()

The following code fragment uses an Enumeration interface to examine every object in avector:

Enumeration vectEnum = myVector.elements(); // get the vector'senumerationwhile (vectEnum.hasMoreElements()) // while there's something to get...

{

Object nextOb = vectEnum.nextElement(); // get the next object

// do whatever you want with the next object

}

This loop works the same for every data structure that can return an Enumeration object. Adata structure typically has an elements() method, or something similar, that returns theenumeration. After that, the kind of data structure doesn't matter—they all look the samethrough the Enumeration interface.

Searching for Objects in a Vector:

You can always search for objects in a vector manually, by using an enumeration and doing anelement-by-element comparison, but you will save a lot of time by using the built-in searchfunctions.If you just need to know whether an object is present in a vector, use the contains() method.For example:

public final boolean contains(Object ob)

returns true if ob occurs at least once in the vector, or false if not.You can also find out an object's position in a vector with the indexOf() and lastIndexOf() methods.

For example:

public final int indexOf(Object ob)

returns the position in the vector where the first occurrence of ob is found, or -1 if ob is notpresent in the vector.

public final synchronized int indexOf(Object ob, int startIndex)

throws ArrayIndexOutOfBoundsException

returns the position in the vector where the first occurrence of ob is found, starting at positionstartIndex. If ob is not in the vector, it returns -1. If startIndex is less than 0, or greater thanor equal to the vector’s length, you get an ArrayOutOfBoundsException.

public final int lastIndexOf(Object ob)

returns the position in the vector where the last occurrence of ob is found, or -1 if ob is notpresent in the vector.

public final synchronized int lastIndexOf(Object ob, int startIndex)

throws ArrayOutOfBoundsException

returns the position in the vector where the last occurrence of ob is found, starting at positionstartIndex. If ob is not in the vector, it returns -1. If startIndex is less than 0 or greater thanor equal to the vector’s length, you get an ArrayOutOfBoundsException.

Removing Objects from a Vector:

You have three options when it comes to removing objects from a vector. You can remove alltheobjects, remove a specific object, or remove the object at a specific position TheremoveAllElements method removes all the objects from a vector:

public final synchronized void removeAllElements()

The removeElement method removes a specific object from a vector:

public final synchronized boolean removeElement(Object ob)

If the object occurs more than once, only the first occurrence is removed. The method returnstrue if an object was actually removed, or false if the object was not found in the vector.The removeElementAt method removes the object at a specific position and moves the otherobjects over to fill in the gap created by the removed object:

public final synchronized void removeElementAt(int index)

throws ArrayIndexOutOfBoundsException

If you try to remove an object from a position that does not exist, you get anArrayIndexOutOfBoundsException.

Changing the Size of a Vector:

A vector has two notions of size—the number of elements currently stored in the vector andthe maximum capacity of the vector. The capacity method tells you how many objects thevector canhold before having to grow:
public final int capacity()

You can increase the capacity of a vector using the ensureCapacity method. For example:

public final synchronized void ensureCapacity(int minimumCapacity)tells the vector that it should be able to store at least minimumCapacity elements. If the vector'scurrent capacity is less than minimumCapacity, it allocates more space. The vector does notshrink the current capacity if he capacity is already higher than minimumCapacity.If you want to reduce a vector's capacity,use the trimToSize method:
public final synchronized void trimToSize()

This method reduces the capacity of a vector down to the number of elements it is urrentlystoring.The size method tells you how many elements are stored in a vector:
public final int size()

You can use the setSize method to change the current number of elements:

public synchronized final void setSize(int newSize)
If the new size is less than the old size, the elements at the end of the vector are lost. If the newsize is higher than the old size, the new elements are set to null. Calling setSize(0) is the
same as calling removeAllElements().

14.4 THE HASHTABLE CLASS

The Hashtable class is the most common implementation of the Map collection and providesmethods for associating one object with another. Hashtables are often used to associate a namewith an object and retrieve the object based on that name. In a dictionary, the name object iscalled a *key*, and it can be any kind of object. The object associated with the key is called the*value*. A key can be associated with only one value, but a value can have more than one key.Effectively you can think of the Hashtable class as a class that uses the hash codes of the keyobjects to perform the lookup. It groups keys into *buckets* based on their hash code. When itgoes to find a key, it queries the key's hash code, uses the hash code to get the correct bucket,and then searches the bucket for the correct key. Usually, the number of keys in the bucket issmall compared to the total number of keys in the hashtable, so the hashtable performs only afraction of the comparisons performed in most collections like a vector.

The hashtable has a capacity, which tells how many buckets it uses, and a load factor, which isthe ratio of the number of elements in the table to the number of buckets. When you create ahashtable, you can specify a load factor threshold value. When the current load factor exceedsthis threshold, the table grows—that is, it doubles the number of buckets and then reorganizesthe table. The default load factor threshold is 0.75, which means that when the number ofelements stored in the table is 75 percent of the number of buckets, the number of buckets isdoubled. You can specify any load factor threshold greater than 0 and less than or equal to 1. Asmaller threshold means a faster lookup because there will be few keys per bucket (maybe nomore than one), but the table will have far more buckets than elements, so there is somewasted space. A larger threshold means the possibility of slower lookups, but the number ofbuckets is closer to the number of elements.

The Hashtable class has three constructors.

public Hashtable()

creates a new hashtable with a default capacity of 101 and a default load factor threshold of0.75.

public Hashtable(int initialCapacity)

The Hashtable Class :

creates a new hashtable with the specified initial capacity and a default load factor threshold of0.75.

public Hashtable(int initialCapacity, float loadFactorThreshold)

throws IllegalArgumentException

creates a new hashtable with the specified initial capacity and threshold. If the initial capacity is0 or less, or if the threshold is 0 or less, or greater than 1, you get anIllegalArgumentException.

Storing Objects in a Hashtable

To store an object in a dictionary with a specific key, use the put() method:

public abstract Object put(Object key, Object value)

throws NullPointerException

The object returned by the put method is the object previously associated with the key. If therewas no previous association, the method returns null. You cannot have a null key or a nullvalue. If you pass null for either of these parameters, you get a NullPointerException.

Retrieving Objects from a Hashtable

The get() method finds the object in the hashtable associated with a particular key:

public abstract Object get(Object key)

The get() method returns null if there is no value associated with that key.

Removing Objects from a Hashtable

To remove a key-value pair from a dictionary, call the remove() method with the key. For example:

public abstract Object remove(Object key)

returns the object associated with the key, or null if no value is associated with that key.The Dictionary class also provides some utility methods that give you information about thedictionary. The isEmpty() method returns true if no objects are stored in thedictionary:

public abstract boolean isEmpty()

The size method tells you how many key-value pairs are currently stored in the dictionary:

public abstract int size()

The keys method returns an Enumeration object that allows you to examine all the keys in thedictionary, whereas the elements() method returns an Enumeration for all the values in thedictionary:

public abstract Enumeration keys()

public abstract Enumeration elements()

In addition to these methods, the Hashtable has a few more methods for the Map interface.

public synchronized void clear()

removes all the elements from the hashtable. This is similar to the removeAllElements methodin the Vector class.

public synchronized boolean contains(Object value)

throws NullPointerException

returns true if value is stored as a value in the hashtable. If value is null, it throws aNullPointerException.

public synchronized boolean containsKey(Object key)

returns true if key is stored as a key in the hashtable.

When a hashtable grows in size, it has to rearrange all the objects in the table over the new setof buckets. In other words, if there were 512 buckets and the table grew to 1,024 buckets, youneed to redistribute the objects over the full 1,024 buckets. An object’s bucket is determined bya combination of both the hash code and the number of buckets. If you were to change thenumber of buckets but not rearrange the objects, the hashtable might not be able to locate anexisting object because its bucket was determined based on a smaller size. The rehash()method, (which is automatically called when the table grows) recomputes the location of eachobject in the table.

14.5 THE PROPERTIES CLASS

The Properties class is a special kind of dictionary that uses strings for both keys and values.It is used by the System class to store system properties, but you can use it to create your ownset of properties. The Properties class is actually just a hashtable that specializes in storingstrings.

You can create a new Properties object with the empty constructor:

public Properties()

You can also create a Properties object with a set of default properties. When the Propertiesobject cannot find a property in its own table, it searches the default properties table. If youchange a property in your own Properties object, it does not change the property in the defaultProperties object. This means that multiple Properties objects can safely share thesame default Properties object. To create a Properties object with a default set of properties,just pass the default Properties object to the constructor:

public Properties(Properties defaultProps)

Setting Properties

You set properties using the same put() method that all dictionaries use:

public Object put(Object key, Object value)

throws NullPointerException

Querying Properties:

The getProperty() method returns the string corresponding to a property name, or null ifthe property is not set:
public String getProperty(String key)

If you specify a default Properties object, that object is also checked before null is returned.You can also call getProperty and specify a default value to be returned if the property isnot set:

public String getProperty(String key, String defaultValue)
In this version of the getProperty method, the default Properties object is completely ignored.The value returned is either the property corresponding to the key, or, if the property isnot set, defaultValue.

You can get an Enumeration object for all the property names in a Properties object, includingthe default properties, with the propertyNames() method:

public Enumeration propertyNames()

Saving and Retrieving Properties:

Because the Properties class is so useful for storing things like a user’s preferences, you needa way to save the properties to a file and read them back the next time your program starts.You can use the load() and save() methods for this.public synchronized void save(OutputStream out, String header)saves the properties on the output stream out. The header string is written to the stream beforethe contents of the Properties object.public synchronized void load(InputStream in)throws IOExceptionreads properties from the input stream. It treats the # and ! characters as comment charactersand ignores anything after them up to the end of the line, similar to the // comment charactersin Java.

Listing 14.1 shows a sample file written by the save() method.

Listing 14.1 File Written by the save() Method

```
#Example Properties
#Mon Jun 17 19:57:39 1996
foo=bar
favoriteStooge=curly
helloMessage=hello world!
```

The list() method is similar to the save() method, but it presents the properties in a morereadable form. It displays the contents of a properties table on a print stream in a nice, friendlyformat, which is handy for debugging. The format of the list() method is as follows:

```
public void list(PrintStream out)
```

14.6 THE STACK CLASS

A stack is a handy data structure that adds items in a last-in, first-out manner. In other words, when you ask a stack to give you the next item, it hands back the most recently added item.

Think of the stack as a stack of cafeteria trays. The tray on the top of the stack is the last tray you put on the stack. Every time you add another tray it becomes the new top of the stack.

The Stack class is implemented as a subclass of Vector, which means that all the vector methods are available to you in addition to the stack-specific ones. You create a stack with the empty

constructor:

```
public Stack()
```

To add an item to the top of the stack, you push it onto the stack:

```
public Object push(Object newItem)
```

The object returned by the push() method is the same as the newItem object. The

pop()method removes the top item from the stack:

```
public Object pop() throws EmptyStackException
```

If you try to pop an item off an empty stack you get an EmptyStackException. You can find out which item is on top of the stack without removing it by using the peek() method:

```
public Object peek() throws EmptyStackException
```

The empty() method returns true if there are no items on the stack:

```
public boolean empty()
```

Sometimes you may want to find out where an object is in relation to the top of the stack. Because you don’t know exactly how the stack stores items, the indexOf() and lastIndexOf() methods from the Vector class might not do you any good. The search() method, however, tells you how far an object is from the top of the stack:

```
public int search(Object ob)
```

If the object is not on the stack at all, search() returns -1.

The Stack Class

The fragment of code in Listing 14.2 creates an array of strings and then uses a stack to reverse the order of the words by pushing them all on the stack and popping them back off.

Listing 14.2 Example Usage of a Stack

```
String myArray[] = { “Please”, “Reverse”, “These”, “Words” };
Stack myStack = new Stack();
// Push all the elements in the array onto the stack
for (int i=0; i < myArray.length; i++) {
myStack.push(myArray[i]);
}
// Pop the elements off the stack and put them in the
// array starting at the beginning
for (int i=0; i < myArray.length; i++) {
myArray[i] = (String) myStack.pop();
}
// At this point, the words in myArray will be in
// the order: Words, These, Reverse, Please
```

14.7 THE DATE CLASS

The Date class represents a specific date and time. It is centered around the Epoch, which is midnight GMT on January 1, 1970. Although there is some support in the Date class for referencing dates as early as 1900, none of the date methods function properly on dates occurring before the Epoch.

The empty constructor for the Date class creates a Date object from the current time:

public Date() You can also create a Date object using the number of milliseconds from the Epoch, the same kind of value returned by System.currentTimeMillis():

public Date(long millis) You can also get the milliseconds since the Epoch by using the static UTC method in the Date class (UTC stands for Universal Time Coordinates):

```
public static long UTC(int year, int month, int date,
int hours, int minutes, int seconds)
```

The following Date constructors allow you to create a Date object by giving a specific year, month, day, and so on:

```
public Date(int year, int month, int date)
public Date(int year, int month, int date, int hours, int minutes)
public Date(int year, int month, int date, int hours, int minutes, int seconds)
```

There are several important things to note when creating dates this way:

- The year value is the number of years since 1900. For instance, the year value for 1984 would be 84.
- Months are numbered starting at 0, not 1. January is month 0.
- Dates (the day of the month) are numbered starting at 1, just to add some confusion, so the 11th day of the month would have a date value of 11.
- Hours, minutes, and seconds are all numbered starting at 0, which, unlike the months, is correct. An hour value of 1 means 1 a.m.

The Date class also has the capability to create a new Date object from a string representation of a date:

```
public Date(String s)
```

The following statements all create Date objects for January 12, 1992 (the birthday of the HAL 9000 computer):

```
Date d = new Date(“January 12, 1992”);
Date d = new Date(92, 0, 12);
Date d = new Date(695174400000l); // milliseconds since the epoch
Date d = new Date(Date.UTC(92, 0, 12, 0, 0, 0));
```

Whenever you create a date using specific year, month, day, hour, minute, and second values, or when you print out the value of a Date object, it uses the local time zone. The UTC method and the number of milliseconds since the Epoch are always in GMT (Greenwich Mean Time). Comparing Dates

As is true with all subclasses of Object, you can compare two dates with the equals() method.

The Date class also provides methods for determining whether one date comes before or after another. The after() method in a Date object returns true if the date comes after the date passed to the method:

```
public boolean after(Date when)
```

The before() method tells whether a Date object occurs before a specific date:

```
public boolean before(Date when)
```

Suppose you defined date1 and date2 as:

```
Date date1 = new Date(76, 6, 4); // July 4, 1976
Date date2 = new Date(92, 0, 12); // January 12, 1992
```

For these two dates, date1.before(date2) is true, and date1.after(date2) is false.

Converting Dates to Strings:

You can always use the toString() method to convert a date to a string. It converts the date to a string representation using your local time zone. The toLocaleString() method also converts a date to a string representation using the local time zone, but the format of the string is slightly different:

```
public String toLocaleString()
```

The toGMTString() method converts a date to a string using GMT as the time zone:

```
public String toGMTString()
```

The following example shows the formats of the different string conversions. The original time was defined as midnight GMT, January 12, 1992. The local time zone is Eastern Standard, or five hours behind GMT.

```
Sat Jan 11 19:00:00 1992 // toString
01/11/92 19:00:00 // toLocaleString
12 Jan 1992 00:00:00 GMT // toGMTString
```

Changing Date Attributes

You can query and change almost all the parts of a date. The only two things that you can query but not change are the time zone offset and the day of the week a date occurs on. The time zone offset is the number of minutes between the local time zone and GMT. The number is positive if your time zone is behind GMT—that is, if midnight GMT occurs before midnight in your time zone. The format of getTimezoneOffset() is:

```
public int getTimezoneOffset()
```

The getDay() method returns a number between 0 and 6, where 0 is Sunday:

```
public int getDay()
```

Remember that the day is computed using local time.

If you prefer to deal with dates in terms of the raw number of milliseconds since the Epoch, you can use the getTime() and setTime() methods to modify the date:

public long getTime()

public void setTime(long time)

You can also manipulate the individual components of the dates using these methods:

public int getYear()

public int getMonth()

public int getDate()

public int getHours()

public int getMinutes()

public int getSeconds()

public void setYear(int year)

public void setMonth(int month)

public void setDate(int date)

public void setHours(int hours)

public void setMinutes(int minutes)

public void setSeconds(int seconds)

14.8 THE BITSET CLASS

The BitSet class provides a convenient way to perform bitwise operations on a large numberof bits, and to manipulate individual bits. The BitSet automatically grows to handle more bits.You can create an empty bit set with the empty constructor:
public BitSet()

If you have some idea how many bits you will need, you should create the BitSet with a specificsize:public BitSet(int numberOfBits)

Bits are like light switches, they can be either on or off. If a bit is set, it is considered on,whereas it is considered off if it is cleared. Bits are frequently associated with Boolean valuesbecause each has only two possible values. A bit that is set is considered to be true, whereas abit that is cleared is considered to be false.You use the set() and clear() methods to set and clear individual bits in a bit set:

public void set(int whichBit)

public void clear(int whichBit)

If you create a bit set of 200 bits and you try to set bit number 438, the bit set automaticallygrows to contain at least 438 bits. The new bits will all be cleared initially. The size() methodtells you how many bits are in the current bit set:public int size()

You can test to see whether a bit is set or cleared using the get() method:

public boolean get(int whichBit)

The get() method returns true if the specified bit is set, or false if it is cleared.There are three operations you can perform between two bit sets. These operations manipulatethe current bit set using bits from a second bit set. Corresponding bits are matched to performthe operation. In other words, bit 0 in the current bit set is compared to bit 0 in the second bitset. The bitwise operations are:

- The or operation sets the bit in the current bit set if either the current bit or the second
- bit is set. If neither bit is set, the current bit remains cleared.
- The and operation sets the bit in the current bit set only if the current bit and the second
- bit are set. Otherwise, the current bit is cleared.
- The xor operation sets the bit in the current bit set if only one of the two bits is set. If both are set, the current bit is cleared.The format of these bitwise operations is:

public void or(Bitset bits)
public void and(Bitset bits)
public void xor(Bitset bits)

14.9 THE STRINGTOKENIZERCLASS

The StringTokenizer class helps you parse a string by breaking it up into tokens. It recognizes tokens based on a set of delimiters. A token is considered to be a string of characters that are not delimiters. For example, the phrase “I am a sentence” contains a number of tokens with spaces as delimiters. The tokens are I, am, a, and sentence. If you were using the colon character as a delimiter, the sentence would be one long token called “I am a sentence” because there are no colons to separate the words. The StringTokenizer is not bound by the convention that words are separated by spaces. If you tell it that words are only separated by colons, it considers spaces to be part of a word.

You can even use a set of delimiters, meaning that many different characters can delimit tokens.

For example, if you had the string “Hello. How are you? I am fine, I think,” you would want to use a space, period, comma, and question mark as delimiters to break the sentence into tokens that are only words.

The string tokenizer doesn't have a concept of words itself; it only understands delimiters.

When you are parsing text, you usually use whitespace as a delimiter. Whitespace consists of spaces, tabs, newlines, and returns. If you do not specify a string of delimiters when you create a string tokenizer, it uses whitespace.

You create a string tokenizer by passing the string to be tokenized to the constructor:

public StringTokenizer(String str)

If you want something other than whitespace as a delimiter, you can also pass a string containing the delimiters you want to use:

public StringTokenizer(String str, String delimiters)
Sometimes you want to know what delimiter is used to separate two tokens. You can ask the string tokenizer to pass delimiters back as tokens by passing true for the returnTokens parameter in this constructor:

public StringTokenizer(String str, String delimiters, boolean returnTokens)

The nextToken() method returns the next token in the string:
public String nextToken() throws NoSuchElementExceptionIf there are no more tokens, it throws a NoSuchElementException. You can use thehasMoreTokens() method to determine whether there are more tokens before you usenextToken():
public boolean hasMoreTokens()

You can also change the set of delimiters on-the-fly by passing a new set of delimiters to the nextToken() method:
public String nextToken(String newDelimiters)

The new delimiters take effect before the next token is parsed and stay in effect until they are changed again.

The countTokens() method tells you how many tokens are in the string, assuming that the delimiter set doesn't change:
public int countTokens()

You may have noticed that the nextToken() and hasMoreTokens() methods look similar to the nextElement() and hasMoreElements() methods in the Enumeration interface. They are so similar, in fact, that the StringTokenizer also implements an Enumeration interface that is implemented as:

```
public boolean hasMoreElements() {
return hasMoreTokens();
}
public Object nextElement() {
return nextToken();
}
```

The following code fragment prints out the words in a sentence using a string tokenizer:

```
String sentence = “This is a sentence”;
StringTokenizer tokenizer = new StringTokenizer(sentence);
while (tokenizer.hasMoreTokens())
{
System.out.println(tokenizer.nextToken());
}
```

14.10 THE RANDOM CLASS

The Random class provides a random number generator that is more flexible than the randomnumber generator in the Math class. Actually, the random number generator in the Math classjust uses one of the methods in the Random class. Because the methods in the Random class arenot static, you must create an instance of Random before you generate numbers. The easiestway to do this is with the empty constructor:

```
public Random()
```

One handy feature of the Random class is that it lets you set the random number seed that eterminesthe pattern of random numbers. Although you cannot easily predict what numbers willbe generated with a particular seed, you can duplicate a series of random numbers by usingthe same seed. In other words, if you create an instance of Random with the same seed valueevery time, you will get the same sequence of random numbers every time. This might not begood for writing games and would be financially devastating for lotteries, but it is useful whenwriting simulations where you want to replay the same sequences over and over. The emptyconstructor uses System.currentTimeMillis to seed the random number generator. To createan instance of Random with a particular seed, just pass the seed value to the constructor:

```
public Random(long seed)
```

You can change the seed of the random number generator at any time using the setSeed()method:

```
        public synchronized void setSeed(long newSeed)
```

The Random class can generate random numbers in four different data types.

public int nextInt()

generates a 32-bit random number that can be any legal int value.

public long nextLong()

generates a 64-bit random number that can be any legal long value.

public float nextFloat()

generates a random float value between 0.0 and 1.0, though always less than 1.0.

public double nextDouble()

generates a random double value between 0.0 and 1.0, always less than 1.0. This is the methodused by the Math.random() method.There is also a special variation of random number that has some interesting mathematicalproperties. This variation is called nextGaussian.public synchronized double nextGaussian()returns a special random double value that can be any legal double value. The mean (average)of the values generated by this method is 0.0, and the standard deviation is 1.0. This means thatthe numbers generated by this method are usually close to zero and that very large numbersare fairly rare.

14.11 THE OBSERVABLE CLASS

The Observable class allows an object to notify other objects when it changes. The concept ofobservables is borrowed from Smalltalk. In Smalltalk, an object may express interest in another object, meaning that it would like to know when the other object changes.When building user interfaces, you might have multiple ways to change a piece of data, andchanging that data might cause several different parts of the display to update. For instance,suppose that you want to create a scrollbar that changes an integer value and, in turn, thatinteger value is displayed on some sort of graphical meter. You want the meter to update as thevalue is changed, but you don't want the meter to know anything about the scrollbar. If you arewondering why the meter shouldn't know about the scrollbar, what happens if you decide youdon't want a scrollbar but want the number entered from a text field instead? You shouldn'thave to change the meter every time you change the input source.You would be better off creating an integer variable that is observable. It allows other objects toexpress interest in it. When this integer variable changes, it notifies those interested parties(called observers) that it has changed. In the case of the graphical meter, it would be informedthat the value changed and would query the integer variable for the new value and then redrawitself. This allows the meter to display the value correctly no matter what you are using tochange the value.This concept is known as Model-View-Controller. A model is the nonvisual part of an application.

In the preceding example, the model is a single integer variable. The view is anything thatvisually displays some part of the model. The graphical meter is an example of a view. Thescrollbar could also be an example of a view because it updates its position whenever the integervalue changes. A controller is any input source that modifies the view. The scrollbar, in thiscase, is also a controller (it can be both a view and a controller).In Smalltalk, the mechanism for expressing interest in an object is built right in to the Objectclass. Unfortunately, for whatever reason, Sun separated out the observing mechanism into aseparate class. This means extra work for you because you cannot just register interest in anInteger class; you must create your own subclass of Observable.The most important methods to you in creating a subclass of Observable are setChanged()and notifyObservers(). The setChanged() method marks the observable as having beenchanged, so that when you call notifyObservers() the observers are notified:

protected synchronized void setChanged()The setChanged() method sets an internal changed flag that is used by the notifyObservers()method. It is automatically cleared when notifyObservers() is called, but you can clear itmanually with the clearChanged() method:
protected synchronized void clearChanged()

The notifyObservers() method checks to see whether the changed flag has been set, and ifnot, it does not send any notification:public void notifyObservers()

The following code fragment sets the changed flag and notifies the observers of the change:

```
setChanged(); // Flag this observable as changed
notifyObservers(); // Tell observers about the change
```

The notifyObservers() method can also be called with an argument:

public void notifyObservers(Object arg)

This argument can be used to pass additional information about the change—for instance, thenew value. Calling notifyObservers() with no argument is equivalent to calling it with anargument of null.You can determine whether an observable has changed by calling the hasChanged() method:
public synchronized boolean hasChanged()

Observers can register interest in an observable by calling the addObserver() method:

```
public synchronized void addObserver(Observer obs)
```

Observers can deregister interest in an observable by calling deleteObserver():

```
public synchronized void deleteObserver(Observer obs)
```

An observable can clear out its list of observers by calling the deleteObservers() method:

```
public synchronized void deleteObservers()
```

The countObservers() method returns the number of observers registered for an observable:

```
public synchronized int countObservers()
```

Listing 14.3 shows an example implementation of an ObservableInt class.

Listing 14.3 Source Code for ObservableInt.java

```
import java.util.*;
// ObservableInt - an integer Observable
//
// This class implements the Observable mechanism for
// a simple int variable.
// You can set the value with setValue(int)
// and int getValue() returns the current value.
public class ObservableInt extends Observable
{
int value; // The value everyone wants to observe
public ObservableInt()
{
value = 0; // By default, let value be 0
}
public ObservableInt(int newValue)
{
value = newValue; // Allow value to be set when created
}
public synchronized void setValue(int newValue)
{
//
// Check to see that this call is REALLY changing the value
//
if (newValue != value)
{
value = newValue;
setChanged(); // Mark this class as “changed”
notifyObservers(); // Tell the observers about it
}
}
public synchronized int getValue()
{
return value;
}
}
```

The Observable class has a companion interface called Observer. Any class that wants to receiveupdates about a change in an observable needs to implement the Observer interface. The Observer interface consists of a single method called update() that is called when an objectchanges. The format of update() is:

public abstract void update(Observable obs, Object arg);
where obs is the observable that has just changed, and arg is a value passed by the observablewhen it called notifyObservers(). If notifyObservers() is called with no arguments, arg isnull.

Listing 14.4 shows an example of a Label class that implements the Observer interface so thatit can be informed of changes in an integer variable and update itself with the new value.Listing 14.4 Source Code for IntLabel.java

```
import java.awt.*;
import java.util.*;
//
// IntLabel - a Label that displays the value of
// an ObservableInt.
public class IntLabel extends Label implements Observer
{
private ObservableInt intValue; // The value we’re observing
public IntLabel(ObservableInt theInt)
{
intValue = theInt;
// Tell intValue we’re interested in it
intValue.addObserver(this);
// Initialize the label to the current value of intValue
setText(“”+intValue.getValue());
}
// Update will be called whenever intValue is changed, so just update
// the label text.
public void update(Observable obs, Object arg)
{
setText(“”+intValue.getValue());
}
}
```

Now that you have a model object defined in the form of the ObservableInt and a view in theform of the IntLabel, you can create a controller—the IntScrollbar. Listing 14.5 shows theimplementation of IntScrollbar.

Listing 14.5 Source Code for IntScrollbar.java

```
import java.awt.*;
import java.util.*;
//
// IntScrollbar - a Scrollbar that modifies an
// ObservableInt. This class functions as both a
// “view” of the observable, since the position of
// the scrollbar is changed as the observable’s value
// is changed, and it is a “controller,” since it also
// sets the value of the observable.
//
// IntScrollbar has the same constructors as Scrollbar,
// except that in each case, there is an additional
// parameter that is the ObservableInt.
// Note: On the constructor where you pass in the initial
// scrollbar position, the position is ignored.
public class IntScrollbar extends Scrollbar implements Observer
{
private ObservableInt intValue;
// The bulk of this class is implementing the various
// constructors that are available in the Scrollbar class.
public IntScrollbar(ObservableInt newValue)
{
super(); // Call the Scrollbar constructor
intValue = newValue;
intValue.addObserver(this); // Register interest
setValue(intValue.getValue()); // Change scrollbar position
}
public IntScrollbar(ObservableInt newValue, int orientation)
{
super(orientation); // Call the Scrollbar constructor
intValue = newValue;
intValue.addObserver(this); // Register interest
setValue(intValue.getValue()); // Change scrollbar position
}
public IntScrollbar(ObservableInt newValue, int orientation,
int value, int pageSize, int lowValue, int highValue)
{
super(orientation, value, pageSize, lowValue, highValue);
intValue = newValue;
intValue.addObserver(this); // Register interest
setValue(intValue.getValue()); // Change scrollbar position
}
// The handleEvent method checks with the parent class (Scrollbar) to see
// if it wants the event, if not, just assumes the scrollbar value has
// changed and updates the observable int with the new position.
public boolean handleEvent(Event evt)
{
if (super.handleEvent(evt))
{
return true; // The Scrollbar class handled it
}
intValue.setValue(getValue()); // Update the observable int
return true;
}
// update is called whenever the observable int changes its value
public void update(Observable obs, Object arg)
{
setValue(intValue.getValue());
}
}
```

This may look like a lot of work, but watch how easy it is to create an applet with anIntScrollbar that modifies an ObservableInt and an IntLabel that displays one. Listing 14.6shows an implementation of an applet that uses the IntScrollbar, the ObservableInt, and theIntLabel.

Listing 14.6 Source Code for ObservableApplet1.java

```
import java.applet.*;
import java.awt.*;
public class ObservableApplet1 extends Applet
{
ObservableInt myIntValue;
public void init()
{
// Create the Observable int to play with
myIntValue = new ObservableInt(5);
setLayout(new GridLayout(2, 0));
// Create an IntScrollbar that modifies the observable int
add(new IntScrollbar(myIntValue,
Scrollbar.HORIZONTAL,
0, 10, 0, 100));
// Create an IntLabel that displays the observable int
add(new IntLabel(myIntValue));
}
}
```

You might notice when you run this applet that the label value changes whenever you updatethe scrollbar; yet the label has no knowledge of the scrollbar, and the scrollbar has no knowledgeof the label.Now, suppose that you also want to allow the value to be updated from a TextField. All youneed to do is create a subclass of TextField that modifies the ObservableInt. Listing 14.7shows an implementation of an IntTextField.

Listing 14.7 Source Code for IntTextField.java

```
import java.awt.*;
import java.util.*;
//
// IntTextField - a TextField that reads in integer values and
// updates an Observable int with the new value. This class
// is both a “view” of the Observable int, since it displays
// its current value, and a “controller” since it updates the
// value.
public class IntTextField extends TextField implements Observer
{
private ObservableInt intValue;
public IntTextField(ObservableInt theInt)
{
// Initialize the field to the current value, allow 3 input columns
super(“”+theInt.getValue(), 3);
intValue = theInt;
intValue.addObserver(this); // Express interest in value
}
// The action for the text field is called whenever someone presses “return”
// We’ll try to convert the string in the field to an integer, and if
// successful, update the observable int.
public boolean action(Event evt, Object whatAction)
{
Integer intStr; // to be converted from a string
try { // The conversion can throw an exception
intStr = new Integer(getText());
// If we get here, there was no exception, update the observable
intValue.setValue(intStr.intValue());
} catch (Exception oops) {
```

```
// We just ignore the exception
}
return true;
}
// The update action is called whenever the observable int’s value changes.
// We just update the text in the field with the new int value
public void update(Observable obs, Object arg)
{
setText(“”+intValue.getValue());
}
}
```

After you have created this class, how much code do you think you have to add to the applet?

You add one line (and change GridLayout to have three rows). Listing 14.8 shows an implementation of an applet that uses an ObservableInt, an IntScrollbar, an IntLabel, and an IntTextField.

Listing 14.8 Source Code for ObservableApplet2.java

```
import java.applet.*;
import java.awt.*;
public class ObservableApplet2 extends Applet
{
ObservableInt myIntValue;
public void init()
{
// Create the Observable int to play with
myIntValue = new ObservableInt(5);
setLayout(new GridLayout(3, 0));
// Create an IntScrollbar that modifies the observable int
add(new IntScrollbar(myIntValue,Scrollbar.HORIZONTAL,0, 10, 0, 100));
// Create an IntLabel that displays the observable int
add(new IntLabel(myIntValue));
// Create an IntTextField that displays and updates the observable int
add(new IntTextField(myIntValue));
}
}
```

Again, the components that modify and display the integer value have no knowledge of each other; yet whenever the value is changed, they are all updated with the new value.

14.12 JAVA VERSUS JAVASCRIPT

JavaScript and Java Integration:

If you’re reading this book from beginning to end, by now you should have a pretty good ideaof what Java is and what it can do for you. Now that you are approaching the end of this book, it is time to introduce you to another language you will encounter if you plan to build Java appletson the Web. The next few chapters bring you to a greater understanding of the features ofNetscape’s new scripting language, JavaScript. You learn how it differs from Sun’s Java andhow you can use JavaScript to dramatically enhance your Web pages. You see that JavaScriptand Java are distinct languages that can work together in a Web browser environment to createpages that are highly interactive.

Programmers often are confused by the similar names of Java and JavaScript. If you were tosay to a friend, “I program in JavaScript,” more often than not, that person will respond withsomething like, “Oh, perhaps then you can help me with this Java applet….”It is a common misconception that Java and JavaScript are just part of the same language. Thisis far from true. Although they are similarly named, there are quite a few differences betweenthem, the first of which is their origin. Around June of 1991, Java was developed at SunMicrosystems and was originally called Oak. It was officially announced (after much developmentand a name change) in May 1995 at SunWorld ’95. JavaScript was developed at NetscapeCommunications Corporation and was originally called LiveScript. Sun renamed Oak to Javabecause of copyright issues with another language already called Oak, and Netscape changedLiveScript to JavaScript after an agreement with Sun to develop JavaScript as a language fornon-programmers. JavaScript was first released with Netscape 2.0.Before you delve into some of the differences between these two languages, let’s get an overviewof the major distinctive points and then discuss each in more depth. Table 54.1 lists someof the major distinctions between Java and JavaScript.

Table 54.1 JavaScript and Java Comparison

JavaScript	**Java**
Developed by Netscape.	Developed by Sun.
Code is interpreted by client	Code is compiled and placed on server before(Web browser). execution on client.
Object-based. Objects are built in but	Object-oriented. Everything is an extensible are not classes and cannot

	use class that can use inheritance.
inheritance.	
Data types need not be declared	Data types must be declared (strong (loose typing).typing).
Runtime check of object references	Compile-time check of object references(static binding). (dynamic binding).
Restricted disk access (must ask before	Restricted disk access (levels of access set by writing a file). user; cannot automatically write to disk).

14.13 JAVASCRIPT IS NOT JAVA

Scripts are limited to Web browser Compiled code can run either as a Web appletfunctionality. or a standalone application.

Scripts work with HTML elements Can handle many kinds of elements (such as(tags). audio and video).
The language is rapidly evolving and Most major changes are completechanging in functionality.

There are few libraries of standard Java comes with many libraries bundled withcode with which to build Web. the language.

The first thing you need to know is that JavaScript is not Java. It is almost becoming a mantrafor JavaScript programmers who constantly face Java questions, even though the forum (suchas the newsgroup) is clearly JavaScript.

You can write JavaScript scripts and never use a single Java applet. The reason JavaScript hasadopted Java's name (in addition to the agreement with Sun Microsystems) is because thelanguage has a similar syntax to Java. Netscape also recognized the momentum building behindJava and leveraged the name to strengthen JavaScript. If you have programmed in Java,then you will find that JavaScript is both intuitive and easy for you to pick up. There is notmuch new for you to learn. The nicest thing about JavaScript, though, is that you don't need tohave any experience using Java to rapidly create useful scripts.To give you an example of the similar nomenclature in Java and JavaScript, look at how eachlanguage would handle a specific function called from a specific object.Suppose in Java you have a class called MyClass that contains a method called MyMethod. Youcould call MyMethod in this way:

```
foo = new MyClass();
result = foo.MyMethod(parameter1, parameter2);
```

In JavaScript, you can do the same thing. If you have a function called MyObject defined by:

```
function MyObject(parameter) {
this.firstone = parameter;
this.MyFunction = MyFunction;
}
```

assuming that MyFunction() has been previously defined, you can then call MyFunction by:

```
foo = new MyObject(parameter);
foo.MyFunction(someparameter);
```

In the first part, you have created two properties (basically slots for information inside theobject) called firstone and MyFunction. In the second part, you see how you can create aspecific instance of an object and use that new object’s methods.

JavaScript Java :

There are many similarities like this between the languages. See the next chapter for details ofthe JavaScript syntax.

Interpreted Versus Compiled

JavaScript code is almost always placed within the HTML document where it will be running.When you load a page that contains JavaScript code, the Web browser contains a built-ininterpreterthat takes the code as it is loaded and executes the instructions (sometimes on-the-fly,before you see anything on that window). This means that you can usually use View Source(choose View, Document Source in the Netscape menu) to see the code inside the HTMLdocument.

JavaScript uses the <script>...</script> tag, similar to Java’s <applet>...</applet> tag.

Everything within the <script>...</script> is ignored by Netscape’s HTML parser, but is

passed on to the JavaScript interpreter. For Web browsers that do not support the

<script>...</script> tag, it is customary to further enclose the JavaScript code in comments.

Here is an example in Listing.14.9.

Listing.14.9 JavaScript Code in an HTML Document

```
<html>
<head>
<title>JavaScript Hello!</title>
<script language=”JavaScript”>
<!-- // to hide from old browsers
var textData = “Hello World!”;
```

```
function showWorld(textInput) {
document.write(textInput);
}
//to ignore end comment -->
</script>
</head>
<body bgcolor=white>
<script language=”JavaScript”>
<!--
showWorld(textData);
// -->
</script>
</body>
</html>
```

The script in Listing.14.9 shows how JavaScript can appear in both the <head> and <body>elements of an HTML document. This script first loads the function within the <head> element.

(All of the code is read from the top down—remember this when you refer to other pieces ofcode so you don’t refer to code that hasn’t been loaded yet.) When the browser encounters theshowWorld(textData) line in the <body> element, the browser displays the text value oftextData—in this case, “Hello World!”.

14.14 INTERPRETED VERSUS COMPILED

If you were to do something similar in Java, you would write the code in Java in an editor, compilethe code to a .class file, and place that file on your server. You would then use the now familiar <applet>...</applet> to embed this applet in your HTML document. For example,the Java code would appear as:

```
public class HelloWorld extends java.applet.Applet {
public static void main (String args []) {
System.out.println(“Hello World”);
}
}
```

The HTML code would appear as:

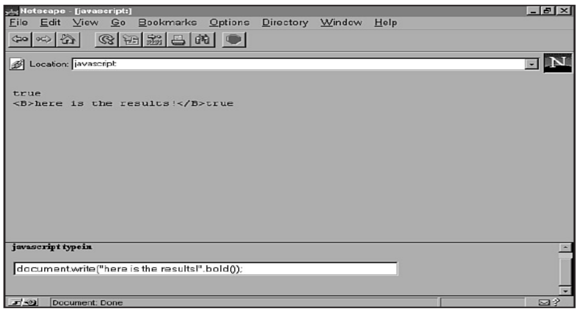
```
<html>
<head>
<title>Java Example</title>
</head>
<body>
```

```
<applet code=”HelloWorld.class” width=150 height=25>
</applet>
</body>
</html>
```

The benefits of having code interpreted by the browser instead of compiled by a compiler andrun through the browser is primarily that you—as a JavaScript developer—can very quicklymake modifications to your code and test the results via the browser. If you use Java, you mustchange the code, compile it, upload it again (if you are testing on your own Web server), andthen view it in your browser. Interpreted code is typically not as fast as compiled code, but forthe limited scope of JavaScript, you will probably see scripts run a little faster than their Javacounterparts (with equivalent functions, such as a scrolling text ticker).The drawback to having your JavaScript code on the HTML document is that any code youwrite will be exposed to anyone else who accesses your page—even those who want to useyour code for their own projects. For small scripts, this is not much of a problem, nor is it aproblem for large projects—if you don’t mind having your efforts used on other pages or improvedupon by others. If you have a large project in which you want to keep your code private,you might consider using Java instead. With the recent implementation of the SRC attribute,your scripts can now be pulled out of the HTML page and placed in their own file. This dramaticallyincreases their usefulness if you have scripts that you want to reuse often. Overall, it ismore convenient for JavaScript coders to be able to see the results of their changes on-the-fly inthe browser than it is for the code/compile/upload/view of Java.One feature of this interpreted nature of JavaScript is that you can test out statements on-the-fly(such as eval(7*45/6) or document.write(“hi1”)). If you are using Netscape Navigator 2.0 orlater, try typing **javascript:** or **mocha:** in the URL window. You see that the browser windowchanges into interpreter mode with a large frame above a smaller frame. You can type inJavaScript commands in the smaller frame input box and see the results displayed in the largerframe (see Figure 54.1).

FIG. 54.1

The JavaScript Interpreter evaluates statements you type in the lower window and displays the results inthe upper window. (Mac and Windows versions vary in their display.)



Location: javascript

```
true
<B>here is the results!</B>true
```

```
document.write("here is the results!".bold());
```

14.15 OBJECT BASED VERSUS OBJECT ORIENTED

JavaScript takes liberally from Java in respect to its overall language structure, but lacks manyof the features that make Java an object-oriented language. JavaScript has built-in objects (suchas Navigator, Window, or Date) that access many browser elements such as windows, links, andimages. Java typically cannot access any of these browser elements and is restricted to the area(or bounding box) that contains it and any Java windows it subsequently creates.JavaScript allows you to create new objects that are really functions. Objects in JavaScript arenot true objects because they do not implement inheritance and other features that Java does.For instance, you cannot create a new class MyWindow that inherits properties of the JavaScriptobject window (the top-level object in JavaScript).Although this limitation may at first seem very constricting, you can still create many usefulfunctions within JavaScript that can perform many of the same tasks that an equivalent Javaapplet can do.Due to these built-in objects, JavaScript really shines when it comes to accessing or manipulatingbrowser-based attributes, such as the current time, the value of a given form element, thethird window in your browser, the link you visited five clicks ago, and so on.Java code can be written to allow a programmer to do just about anything on a computer as astandalone application or a Java applet. But, JavaScript fills a major gap (at least in the contextof Java applets and Web browsers) by acting as a glue by which Java and the browser can communicate.Via JavaScript, you could enter information into a form field in an HTML document,and a Java applet on that page could use that input to display new information. JavaScript allows..

14.16 STRONG TYPING VERSUS LOOSE TYPING

When you are writing JavaScript code, variables don’t need to have data types when they are declared. This loose typing means that it is much easier for JavaScript writers to work on creating and manipulating variables without worrying if the data was an int, float, and so on.

In Java, you must explicitly declare what data type a variable will be before you use it. This is called strong typing and contributes to the overall stability of Java code; but it can cause problems for new programmers. JavaScript allows you to ignore this and assumes that the type of value you first assign to a variable is the type you intended it to be. For instance, if you had a variable HouseType and you assigned it a value of “Victorian,” JavaScript assumes you meant HouseType to be a string all along and does not complain if you did not specifically set HouseType to a string. However, if you had first assigned a value of 4 to HouseType, JavaScript now assumes it is of type INT. Overall, this makes for faster and easier script writing and eliminates needless debugging for variable declarations. However, you must be careful to assign the correct type of value to a variable. This loose typing demonstrates one of the areas in which JavaScript seeks to simplify the process of writing code. Because JavaScript is directed toward non-programmers or minimal programmers, the developers sought to simplify the language in as many ways as possible while still keeping much of the flexibility. Other examples of loose-typed language include HyperTalk, dBASE, and AppleScript.

14.17 DYNAMIC VERSUS STATIC BINDING

Java applets to gain access to properties of an HTML page and allows non-programmers accessto various parts of a Java applet—such as public variables.Although JavaScript does not allow inheritance, there is an interesting new feature called prototype.Prototype allows you to add new properties to any object that you created (with the newstatement) and even add new properties to existing built-in objects. What this means is you canextend existing instances of objects even after they have been defined.For example, you have an object House that has the properties of Light, GarageDoor, andBurglarAlarm. You might already have an instance of this called myHouse. But now you want toextend House by adding ChimneySmoke. Instead of re-defining all of your objects, you can useHouse.prototype.ChimneySmoke = ChimneySmoke.

Now the instance called myHouse can access this new property ChimneySmoke by usingmyHouse.ChimneySmoke.Strong Typing Versus Loose TypingWhen you are writing JavaScript code, variables don’t need to have data types when they aredeclared. This loose typing means that it is much easier for JavaScript writers to work on creatingand manipulating variables without worrying if the data was an int, float, and so on.In Java, you must explicitly declare what data type a variable will be before you use it. This iscalled strong typing and contributes to the overall stability of Java code; but it can cause problemsfor new programmers. JavaScript allows you to ignore this and assumes that the type ofvalue you first assign to a variable is the type you intended it to be. For instance, if you had avariable HouseType and you assigned it a value of “Victorian,” JavaScript assumes you meantHouseType to be a string all along and does not complain if you did not specifically setHouseType to a string. However, if you had first assigned a value of 4 to HouseType, JavaScriptnow assumes it is of type INT. Overall, this makes for faster and easier script writing and eliminatesneedless debugging for variable declarations. However, you must be careful to assign thecorrect type of value to a variable.This loose typing demonstrates one of the areas in which JavaScript seeks to simplify the processof writing code. Because JavaScript is directed toward non-programmers or minimal programmers,the developers sought to simplify the language in as many ways as possible whilestill keeping much of the flexibility. Other examples of loose-typed language includeHyperTalk, dBASE, and AppleScript.

Dynamic Versus Static Binding:

Because JavaScript is interpreted on the client's browser, object references are checked on-thefly as opposed to Java's static binding at compile time. Binding simply means that a variable name is bound to a type—either statically through an explicit declaration of a type with a variable, or dynamically through an implicit association determined by the computer at compile or runtime. Because of JavaScript's dynamic nature, objects in JavaScript can be created on-the-fly as well, and might change the functionality of the script due to some outside factor (time, customer responses, and so on). The Date object is often used to get information about the Web browser's current date, time, day of the year, and more. This object is created at the time theJavaScript code is interpreted in order to get the correct information.

Java programs—with static binding—are typically more stable, because the entire process of compiling the code has already been completed via the Java compiler. Any bad or missing object references have been corrected. This is an advantage when you want the given application to load and run quickly on the user's machine.

14.18 RESTRICTED DISK ACCESS

Security is a hot issue in today's Internet and intranet Web industry for many good reasons.One of the greatest fears people have when they use the Web (or other Internet applicationssuch as e-mail) is that a hostile program will enter their computer and damage or compromisetheir sensitive data. Java has a comprehensive way of dealing with security that allows it to douseful things on your computer while keeping it isolated from your sensitive documents. Javaapplets typically cannot write to your hard drive at all, or if they do, it is in some extremelylimited way.

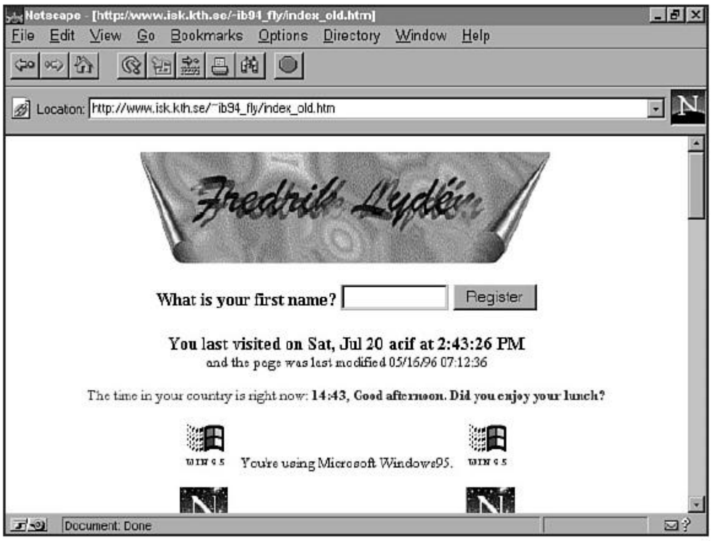
Because JavaScript can control so many aspects of your browser, for a while there was someconcern that JavaScript was less safe than Java. This was primarily due to bugs in early versions of JavaScript that allowed it to send the contents of the file containing your bookmarksand e-mail address to another remote site through a hidden form, or acquire a list of all the fileson your machine. These problems have been eliminated, and JavaScript is now—for the mostpart—as safe as any Java

applet you might run via your browser.Note that JavaScript can write to your hard drive—an essential feature your Web browser hasto write to its cache or save files downloaded via the Web. However, it requires now that youspecifically click Accept in a dialog box to download a file. In a sense, JavaScript is more versatilethan Java in this respect, because it allows you to use your browser to create files to saveand work with at a later time.
If you are concerned that some kind of hostile code might damage your machine, you shouldbe aware that the possibility of virus infection has been around for a long time. Java, with itsassertion of security, has only focused more attention to security issues. Soon, with JavaScript'stainting (a system of marking data so that it cannot be sent via a form or mailto: link viaJavaScript) and Java's digital signatures (an electronic verification of the origin or identity ofthe code or information), you will be able to verify that any code—be it Java or JavaScript—comes from some trusted source. This is very good, because you will be able to allow Java toperform more sensitive tasks such as update your Oracle database or send and auto-installupdated versions of software on your machine.Also note that your Web browser (and JavaScript through the browser) can also write informationto a file called a cookie. This is a file on your machine that allows Web sites to store informationabout you that could be retrieved when you visit that site again. Only the site that wrotethe information to the cookie can retrieve it, and it is usually only used to maintain some kindDifferent Functionality (Scope Limitations) and Code Integration with HTMLof temporary information about you to carry across different pages (such as a user ID or thefact that you are from Florida).

14.19 DIFFERENT FUNCTIONALITY (SCOPE LIMITATIONS) AND CODE INTEGRATION WITH HTML

JavaScript is limited in scope to your Web browser (either Netscape Navigator or MicrosoftInternet Explorer). Java, on the other hand, can run as a standalone application (like Microsoft Word) or within the context of a browser as an applet. It is a testimony of the versatility of Javathat it has adapted so quickly to the Web. It was originally intended to run as operating systemand controls on set-top boxes or other small communication appliances. Given that Java willeventually outstrip C++ as the programming language of choice (at least for Internet programmers,if not for all programmers, as its speed increases), it has the functionality and versatilityto run in many different operating systems.

JavaScript is a smaller language for a more limited audience of Web browser programmers.JavaScript gives to Web programmers the ability to access and modify all of the HTML tags,form elements, window elements, images, bookmarks, links, and anchors in a Web browser. Itenables the programmer to create Web sites that respond and change based on many factorssuch as the time of day or some user profile (see Figure 54.2). JavaScript allows Java applets,scripts, and browser plug-ins to communicate with each other. This actually is a suite of technologiescalled LiveConnect from Netscape and requires some additional code in the Javaapplet or plug-in to be “aware” of JavaScript.


Netscape - [http://www.isk.kth.se/-ib94_fly/index_old.htm]
File
Edit
View
Go
Bookmarks
Options
Directory
Window
Help
Location: http://www.isk.kth.se/~ib94_fly/index_old.htm
Fredrik Lydén
Register
WIN 95
WIN 95
Document: Done

What is your first name?

You last visited on Sat, Jul 20 acif at 2:43:26 PM
and the page was last modified 05/16/96 07:12:36

The time in your country is right now: 14:43, Good afternoon. Did you enjoy your lunch?

You're using Microsoft Windows95.

JavaScript can create HTML files on-the-fly, change attributes of a page instantly(such as thebackground color), and allows the client machine to perform many functions that were traditionallyallowed only through CGIs, such as a TicTacToe game. JavaScript allows for form inputvalidation, where incorrect responses are checked before they are sent back to the browser,which is much more difficult to do in Java than in JavaScript. Essentially, to get the functionalityof JavaScript in Java, you would have to rebuild a mini-browser into your code—a fairlyinefficient solution.Also, JavaScript can be integrated directly into the HTML of your document. Event handlers,such as onClick, can modify the behavior of your browser beyond just accessing new documents.You can create simple calculators in JavaScript that take advantage of the GUI alreadypresent in the browser, as well as the layout capabilities already in place via the presence ofHTML forms or TABLE elements. In other words, you can use your browser to do more thanjust access documents. You can use it as a front end to just about any kind of application. If youuse Java, you have GUI capabilities, but you have to manually activate and resolve these capabilities,which might require significant programming on your part. In JavaScript, though, youlet the browser handle most of the GUI problems and concentrate on your creative project.

Here are some examples of how you can integrate JavaScript statements into your HTML. Yousaw in Listing.1 how you can create scripts via the <script> tag. You can also embedJavaScript statements directly into your HTML:

```
<a href=”” onMouseover=”alert(‘you noticed me!’);”>pass
your mouse over here for a message!</a>
```

Instead of showing the URL in the status bar, passing the mouse over the text of the hyperlinkbrings up a new dialog box that displays the you noticed me! text. To do this in Java, youwould have to create an applet that draws the link text to the screen, write the code that monitorsthe mouse location (probably as a separate thread), and write more code to create anddestroy the resulting dialog box. Needless to say, the JavaScript solution is much easier toimplement for the casual Web designer. Also, with LiveConnect, you can let a Java applet tellJavaScript to open the window and do other tasks without having to write additional code.

Look at the following JavaScript example:

```
<img width=”&{imgwidth}”;%” height=40>
```

Here, JavaScript allows you to set a value to an HTML attribute via a JavaScript expression. Inthis case, if you had defined imgwidth to be 50, the resulting image would have a scaled widthof 50 percent of the window size. Again, if you had to do this in Java, you would have a significantamount of code to create—just to mimic the same ability. Even then, you could not easilyshare the value of imgwidth with other applets.

Overall, the ability to integrate JavaScript code directly in the HTML source allows Web rogrammersto quickly take advantage of the browser’s built-in GUI. Casual scripters can bothleverage their HTML experience as well as any familiarity with Java. Java allows you to createamazing new applications that can be executed on many operating systems that have had theJava interpreter ported to them.

14.20 RAPID EVOLUTION VERSUS RELATIVE STABILITY

The capacity for Java to create as diverse a range of applications as C++ is not disputed. However,if you are a Web page designer who doesn’t have the time to learn how to program Java,you will find that JavaScript is very useful. On the other hand, if you dislike using proprietarycode in your HTML that only functions with two browsers (even though together they holdalmost 90 percent of the current browser market share), you may decide to tough it out byusing just Java.

Rapid Evolution Versus Relative Stability

JavaScript is the newer of the two languages and, as such, is undergoing a more dramatic seriesof changes and improvements. As of this writing, Java Version 1.0.2 is relatively stable.Most of the statements, operators, syntax, and so on have been well-defined and most likelywill not change significantly in subsequent versions. JavaScript has been rapidly evolving fromthe original goal of providing form verification on the client side (instead of a server-side CGI)to having the potential to simulate a simple game such as Pong or breakout. Some of the featuresof JavaScript include:

- Java-JavaScript communication
- JavaScript-Plug-in communication
- Determination of installed plug-ins
- Data Tainting (security enhancements)
- Image reflection (dynamic listing of all images)
- New event handlers (onMouseOut and more)
- New attributes to the <script> tag (SRC)
- New built-in objects (array, string, and so on)
- New operators (typeof)
- Object prototypes

As you can see, JavaScript is changing and growing. It provides a powerful way for nonprogrammers(or light programmers) to do the following:

- Access Java applet methods
- Enable plug-ins and applets to communicate with other elements (other plug-ins, scripts,and applets)
- Validate form information on the client side (before it is sent back to the Web server)
- Generate HTML on-the-fly or based on environment variables (time, date, location, andso on)
- Create simple interactive programs (such as a TicTacToe game) completely in JavaScript.

The drawback to this is subtle. It may seem at first that adding new features with every newrelease of Netscape would be looked upon as a wonderful thing. For the most part this is true,except now when you sit down at your latest browser and begin programming with JavaScript.You have to ask yourself if the feature you are using is going to be available to a large audience(specifically, the target audience for your Web site). Not everyone updates their browser everyfew months, so every release tends to segment your audience. Another drawback is that, forevery feature added to a language, it opens the possibility that a bug snuck in as well.For now, it is a good strategy to take advantage of only those features of JavaScript that havepersisted through Version 2.02 of Netscape. If you use 3.0-specific features, be sure to mentionthis on the page to inform your visitors. So, another difference between Java and

JavaScript isthat, although Java is more powerful and relatively stable as a language, JavaScript is growingwith each version.

It is exciting to see how much a scripter can do now with JavaScript. With 3.0, I have seensimple paint programs, Pong games, and even LED clocks that change every second. Many ofthese scripts would be fairly indistinguishable from their Java counterparts.If you discover a feature you would like to add to JavaScript, you have the uniqueopportunity to talk to the developer of JavaScript (Brendan Eich at Netscape). I expect thatthe expansion of features in JavaScript will continue, especially now that it is being compared toMicrosoft’s Visual Basic Script. If you find JavaScript too limited in some way, you may be able tochange it in future versions. Brendan tries to respond to all email he gets, but with such a high volume,it may take a while for him to respond—if at all.

14.21 LIBRARIES

Sun delivers Java with a standard set of libraries that act to dramatically enhance its usefulness.Instead of having to write all the code to handle images, sockets, and so on, the programmersat Sun have done this for you. You simply have to learn the standard APIs so you can quicklywrite terminal emulators, word processors, and more.JavaScript—because of its relative youth—has not had time to build up any assemblage ofstandard code with which to build Web-based applications. One major problem that stalled thisdevelopment was that you were forced to embed your code in the HTML document in whichyou wanted to use the script. With the addition of the SRC attribute to the <script> tag, youcan now write your code in a separate file and merely reference the script in the page. It issimilar to the CODE attribute in the <applet> tag (this page would load all of the JS files, displaythe correct title at the top of the window, and display a clock above the Welcome to my homepage text

```
<html>
<head>
<script language=”JavaScript” src=”header.js”></script>
</head>
<body>
<script language=”JavaScript”
src=”http://www.foo.com/scripts/timer.js”></script>
<script language=”JavaScript” src=”body.js”></script>
</body>
</html>
```

14.22 JAVASCRIPT AND JAVA INTEGRATION

n HEADER.JS:

document.write(“<title>Welcome!</title>”);

alert(“Welcome To My Homepage!”);

n BODY.JS:

document.write(“Welcome to my home page!”);

The ability to refer to a JavaScript file via the SRC attribute allows you to reuse scripts muchmore readily than before. It is expected to be only a matter of time before standard libraries ofcode are developed and easily accessible. It is somewhat ironic that you can find more standardcode for Java now than you can for JavaScript, given that Java is more complicated.

There are many other examples of differences between Java and JavaScript, such as memoryrequirements (and limitations), threads (Java has them, JavaScript doesn’t), and more. But Ithink that the differences presented in this chapter will help you to perhaps change your perceptionof JavaScript.You should begin to think of JavaScript not as simply an aspect of Java, but instead a complementarylanguage that allows you to greatly control the behavior of your browser. You can nowpass more of the computation and interactivity from your server down to the user’s clientbrowser—thus relieving some of the load and improving the performance of your server.JavaScript is not an all-purpose or a universal scripting language, but in the confines of HTML,plug-ins, browser events, and windows, JavaScript shines as an easy way to add interactivity toyour Web pages. You will see this in the next chapter.

The decision to use Java or JavaScript will depend not only on your skills as a programmer, butalso on the scope of the Web-related task at hand. Look carefully at the task and see if thescrolling text, spinning icon, or calculator might more easily be implemented in JavaScript. Ifyou need to control most of the browser window with specialized text or perhaps have a highlysophisticated application, then Java is surely the way to go.

The next chapter introduces you to the syntax of JavaScript and gives you a good idea of itscapabilities. You may find that programming in JavaScript is as much fun as creating Javaapplets.

14.23 THE BASICS

By now you have read a lot about the newest version of Java, and in the previous chapter, Ibegan to talk about one of

Java’s partners in Web development—JavaScript. You may be wondering(if you jumped straight to this chapter or are reading this in the bookstore) why there isa chapter on JavaScript in a book about Java. The reason is quite simple. JavaScript complementsJava’s capabilities in the Web browser environment. It allows people with little or noprogramming experience who are daunted by Java’s complexity to create interactive and Webbasedapplications.

JavaScript is a scripting language that is loosely based on Java. By imbedding JavaScript codein an HTML document, you can have greater control of your user’s experience as well as pass alarger amount of computation (originally only available via CGI scripts) down to the client-sidebrowser. These scripts are read sequentially by the browser as it is loading a page and canexecute commands immediately—which may affect the page even before it completes loading.Because JavaScript lives inside your HTML document, it can either exist as a complete scriptthat is embedded in the <head> or <body> elements, or it can consist of event handlers that arewritten directly into the HTML code.In Listing.1, you can see how to build the skeleton of a JavaScript script in a document via

the <script> tag.

Listing.1 The Script Tag

```
<SCRIPT LANGUAGE=”JavaScript”>
<!-- HTML comment tags to hide script from old browsers
[JavaScript statements...]
// End hiding the code from old browsers -->
</SCRIPT>
```

You can see from this example that the <script> tag is somewhat similar to the <applet> tagyou use when you embed Java code. The SCRIPT tag has an attribute called LANGUAGE that allowsyou to specify in which language the browser needs to interpret the following code. Thismakes the <script> tag versatile, in that you may eventually use it to embed Visual BasicScript, TCL, Perl, and more scripts.Another attribute to the <script> tag is SRC. Implemented in Netscape 3.0, the SRC attributeallows you to write all of your script in another file and reference that file—instead of having topaste all of the statements in the HTML. If you use the SRC tag, anything you place between the<script>...</script> is ignored. Thus, you can place alternative HTML for non-JavaScriptenabledbrowsers. Listing.2 uses the SRC attribute.

Listing.14.10 JavaScript with SRC

```
<SCRIPT LANGUAGE=”JavaScript” SRC=”footer.js”>
You must not have a JavaScript Enabled Browser if
you see this (poor you!) Click <A HREF=”foo.html”>here</A>
```

```
to go to another page.
</SCRIPT>
```

In Listing.2, the browser loads the script contained within FOOTER.JS as if it had beentyped in the HTML document.

14.24 YOUR FIRST SCRIPT

When you start learning about JavaScript, you will find it very useful to begin with the basics.

Immediately start up your Web browser (Netscape Communicator or Microsoft InternetExplorer)and test out what you learn.Let’s begin with a simple script and explain what will happen when you load this page. Listing55.3 is an example of the typical “Hello World!” program that is very popular for testing out newlanguages. In this example, you are essentially telling the browser to display the string “HelloWorld!” as if you had directly typed that string in your HTML document. (Note that I usuallycapitalize my HTML or JavaScript tags. This is simply a programming convention, but it makesthe code easier to read.)

Listing. 14.11 “Hello World!” Implemented in JavaScript

```
<HTML>
<HEAD>
<SCRIPT LANGUAGE=”JavaScript”>
<!-- Hide me from old browsers
document.write(“Hello World!”);
// End Hiding -->
</SCRIPT>
</HEAD>
<BODY>
Are you ready for JavaScript?
</BODY>
</HTML>
```

Try this one out on your browser and see the results.Basically, the browser reads this code into the JavaScript interpreter. The text string “HelloWorld!” is passed to the write function of the document object, which in turn instructs thebrowser to display the phrase “Hello World!” on a new page. Notice that the actual code is notdisplayed in the browser window. This is because the HTML parser never received this code,as it was passed to the JavaScript interpreter after the HTML parser encountered the <SCRIPT>tag.

14.25 EVENTS

Most of the time, you will be building scripts that do such things as store information, displaydata in a certain format, perform some calculations, or respond to user actions (called events).JavaScript has all of the elements that make up a powerful scripting language and can handleall of these tasks. One of the primary tasks JavaScript is used for is intercepting and handlingevents. Just about any way you respond to your browser can be intercepted by JavaScript.Furthermore, your response can trigger other events, or functions.Essentially, functions are stored chunks of code that are executed at some interval—eitherimmediately, when the document is loaded, or in response to some triggered event. Think offunctions as collections of instructions that allow you to pull out some behavior you might wantto perform over and over again or possibly reuse.When JavaScript encounters an event, it passes it to an event handler. Event handlers are tagsthat point to the specific functions to be executed. Table 14.1 lists the events and handlers inJavaScript.

Table 14.1 Events and Event Handlers in JavaScript

Event	Event Handler	To Trigger Event
blur	onBlur	In a form element, user clicks (tabs) awayfrom element.
click	onClick	In a form element or link, user clickselement.
change	onChange	In a form text, text area, or select object, userchanges value.
focus	onFocus	In a form element, user clicks (tabs) toelement.
load	onLoad	Happens when page is loaded.
mouseover	onMouseOver	Happens when mouse is passed over links oranchors.
select	onSelect	In a form, user selects input field.
submit	onSubmit	In a form, user submits a form (clicks theSubmit button).
unload	onUnload	User leaves the page.

14.26 USING EVENT HANDLERS

Although you can use event handlers anywhere in your JavaScript scripts, you usually placethem either inside HTML form elements or alongside anchors or links. The reason for this isthat JavaScript uses the HTML form as a way to send data to your JavaScript script or performsome “preprocessing” on the data, not just to send data back to the server.For example, an enrollment form on your site asks the users a number of questions aboutthemselves, and you want to make sure they at least fill out their names and ages. BeforeJavaScript, the form was submitted directly back to the Web server, which checked that theappropriate fields were filled out. Then, if they weren’t, the form was sent back to the users,asking for the appropriate information. Now, JavaScript can check this field before it is sent andask the users to fill out that information, without all the overhead of reconnecting to the remoteserver.

Let’s look at an example of how you might add an event handler to your existing HTML code.Most of the time, you follow this general syntax:

```
<TAG eventHandler=”JavaScript code”>
```

Of course, TAG is some HTML tag, and eventHandler is any one of the event handlers you sawin Table 14.1. The “JavaScript code” can be any valid JavaScript code but is usually a call to afunction that you loaded earlier in the document. Listing.14.12 demonstrates an embeddedJavaScript event handler in a common hypertext link. When you click the link, a dialog boxdisplays the text, followed by an OK button for you to click to return to the page. Listing.4 An Event Handler in an HREF

```
<A HREF=”#” onClick=”alert(‘Wow! It Works!’);”>Click here for a message!</A>
```

There is a lot to notice in this example:

n No URL is found in the HREF attribute. Why? You probably don’t want the browser to goto another page while the user is viewing the pop-up window. When the user clicks thelink, not only is the onClick activated, but the browser attempts to go to the locationspecified in the HREF. In this case, you are using this link for its onClick event handlerand not its hypertext reference. An alternative would be to type:

```
<a href=”javascript:alert(‘Wow It Works!’)>Click here</a>
```

n onClick has mixed case. Although HTML is not case sensitive, JavaScript is. This isimportant to remember when you are creating functions and variables.n alert(...) is the standard function for bringing up an alert dialog box on the screen.Notice how this function, and all JavaScript functions, behave similarly to Java in thatthey use parentheses to contain their arguments. In this case, the argument is the string‘Wow! It Works!’. Notice also that the quotation marks of that string are single. Whenyou need to use quotation marks within quotation marks, you nest them by alternatingthe single and double quotation marks. If you need more than two “levels” of quotationsin a given element, you should probably think about an alternate way to eliminate thatneed.

The JavaScript code in quotation marks—”alert(‘Wow! It Works!’);”—ends in asemicolon. You use the semicolon to end a statement in JavaScript, which is similar toPerl and other languages (including Java). Unlike Perl, the use of the semicolon isoptional.

Now that you have seen the two main ways you can implement JavaScript in your HTML code(either in scripts contained by the <SCRIPT>...</SCRIPT> tags or directly embedded in HTMLform elements and links), let’s look at the building blocks of JavaScript code.

14.27 VARIABLES

To create a variable in JavaScript, you simply declare it using the keyword var. You can initializethis variable with some value when you declare it, but it is not required. Listing.14.13 showssome examples of variables created in JavaScript.

Listing.14.13 Variable Declaration in JavaScript

```
var foo = 23
var a, b, c = “letter”
var aNumber = “99”
var isItTrue = false
var flag1 = false , bingo = null , star
```

JavaScript is relatively unique in that you cannot explicitly set a type to a variable, such as castinga string to an integer, like you would in Java. Types are found in JavaScript, but they are setimplicitly. This means that the type a variable has is defined by the context in which it is eitherdefined or used.

When you initialize a variable with a string value (as variables a, b, and c in Listing.14.13), it is astring type; if you initialize it with a number, it becomes an integer type value (as in variable fooin Listing. 14.13).The following places a number of variables within a single statement:bax + bay + baz

This code attempts to treat all of the variables as having the same type as the first variable. Ifbax was a string and bay and baz were originally integers, JavaScript would treat bar and baz asif they were strings. The implicit nature of JavaScript variables allows you to reuse variableseasily without worrying about their type.

If you set some variable day to “Tuesday” and later in the script decide to assign 46 to day, theJavaScript interpreter (inside the browser) will not complain. Because of this, however, youshould be careful when naming your variables so that they do not overlap in scope and causestrange errors in your scripts. You will find it extremely helpful to experiment with declaringand setting variables from the interpreter window I talked about earlier. (In Netscape, just type**javascript:** in the open URL window.)

Table 14.2 contains a list of the possible implicit data types in JavaScript, along with their possiblevalues:

Table 14.2 Data Types in JavaScript

Data Type Values

Number 100, -99.99, 0.000001

Boolean true, false

Strings “this is a string”, “This is another”, “5555”

Null A special keyword with a null value

14.28 VARIABLE NAMES

JavaScript follows the same naming rules for creating variable names as Java. Your variablemust start with a letter or an underscore and can contain subsequent numbers, letters, or underscores. Listing.14.14 gives you a sampling of possible variable names in JavaScript. Rememberto keep your names unique; also remember that, in JavaScript, names are case sensitive.

Listing.14.14 Variable Name Examples

Too_hot

cold999

_100JustRight

This_is_a_long_variable_name_but_it_is_valid000

14.29 VARIABLE SCOPE

Earlier, I mentioned that you want to keep your variable names distinct from one another toprevent overwriting values; but what if you really want to use the same name? This is wherevariable scope comes into play. Global variables are accessible by your entire script and all ofits functions. Local variables are accessible only to the function from which they were created.Those variables are destroyed when that function is complete. To define a variable as a globalvariable, simply assign a value to it (such as “foo = 95”).

To define a local variable inside a function, use the var keyword.

14.30 LITERALS

You can think of a literal as the value on the right side of an equality expression. It is the concreteway to express values in JavaScript and is very similar to Java’s method. Here is a list ofliterals and their possible values:

n Integers:

Decimal expression as a series of digits not starting with a zero:(77, 56565565)

Octal expression as a series of digits starting with a zero:08988

Hexidecimal expression as 0X followed by any digits.

n Floating point:

Expressed as a series of zero or more digits followed by a period (.) and one or moredigits.

Expressed in scientific notation as a series of digits followed by E or e and some digitsfor the exponent (such as -4.666E30).

n Boolean. True or false.

n String. Zero or more characters enclosed by single or double quotation marks.

Strings can contain special characters that affect how they are eventually displayed:

n \b—Backspace

n \f—Linefeed

n \n—New line character

n \r—Carriage return

n \t—Tab character

n \"—An escaped quotation mark—a way to display double quotation marks inside a

string

n \'—Another escaped quotation mark for the single quote

Expressions and Operators

Having values is not enough for a language to be useful. You must have some way to manipulatethese values meaningfully. JavaScript uses expressions to manipulate numbers, strings,and so on. An expression is a set of literals, operators, subexpressions, and variables that evaluateto value. You can use expressions to assign a value to a variable, as in:today = “Friday”Or an expression can simply evaluate to a value, as in :
45 – 66JavaScript uses arithmetical expressions that evaluate to some number, string expressions thatevaluate to another string, and logical expressions that evaluate to true or false. Operatorsbehave very similarly to their cousins in Java. Table 14.3 summarizes the various operators thatare available in JavaScript.

14.31 EXPRESSIONS AND OPERATORS

Table 14.3 JavaScript Operators

Operator Explanation

Computational

+ Numerical addition and string concatenation

- Numerical subtraction and unary negation

* Multiplication

/ Division

% Modulus (remainder)
++ Increment (pre and post)
- - Decrement (pre and post)
Logical
==, !== Equality and inequality (not assignment)
< Less than
<= Less than or equal to
> Greater than
=> Greater than or equal to
! Logical negation (NOT)
&& Logical AND
|| Logical OR
? Trinary conditional selection
, Logical concatenation
Bitwise
& Bitwise AND
| Bitwise OR
^ Bitwise exclusive OR (XOR)
~ Bitwise NOT
<< Left shift
>> Right shift
>>> Unsigned right shift
Assignment
= Assignment
X= Aggregate assignment (where X can be +, -, *,/,%, &, ^, <<, >>, |,
>>>,~) Example: (A += B is equivalent to A = A + B)

The operator precedence is identical to Java's. JavaScript uses lazy evaluation going from left toright. If, while evaluating an expression, it encounters a situation where the expression must befalse, it does not evaluate the rest of the expression and returns false. If you want to groupexpressions to be evaluated first, use the parentheses. For example:

(56 * 99) + (99 - (44 / 5))

A handy expression is the conditional expression. Very underused, this expression allows youto evaluate some condition quickly and return one of two values. Its syntax is:
(condition) ? value1 : value2If the condition is true, then the first value is returned; otherwise, the second is returned. Forexample:isReal = (Imagination <= Reality) true : false

14.32 CONTROL STATEMENTS

Now that you have assignment and mathematical operators, you can assign values to variables,perform simple math expressions, and so on. But you still don’t have the ability to write anykind of meaningful JavaScript code. You need to have some way of controlling the flow of statementevaluation, making decisions based on values, ignoring some statements, and loopingthrough a series of statements until some condition is met.This is where control statements come into play. JavaScript groups these statements into conditional(if...else), loop (for, while, break, continue), object manipulation (for...in, new,this, with), and comments (//, /*...*/). Examples of each of these statements are exploredin this section. (I come back to the object manipulation statements later, after you learn aboutJavaScript’s object model.)JavaScript uses brackets to enclose a series of statements into a complete chunk of code. WhenJavaScript encounters these chunks, all of the statements within are evaluated (unless, ofcourse, JavaScript encounters another branch beforehand, as you learn soon).

Conditional Statements :

These are statements that allow your script to make decisions based on criteria you select.if...else When you want to execute some block of code based on some other condition, youcan use the if statement. Its syntax is:

Table 14.3 Continued

Operator Explanation

Control Statements

```
if ( someExpressionIsTrue) {
zero or more statements...
}
```

If you want to either execute some block of code or another, you can use the if...else statement,which forces the execution of one block or the other. Its syntax is:

```
if ( someExpressionIsTrue) {
some statements...
}
else {
some other statements...
}
```

If you want to execute just one line of code, you can omit the brackets. This is notrecommended, however, because your code will not be as easy to follow later. Listing.14.15 shows how you might implement an if...else statement. It also shows you howyou can chain together multiple else and if statements. Listing. 14.15 if...else Statement Chaining

if...else statement

```
if ( jobs < 100) && (money <= budget) {
poor = true;
free = (99 - x) / jobs ;
}
else if (jobs != overTime) {
workers = “Strike”
}
else {
poor = false;
workers = “Happy”;
}
```

In a moment, I will talk about functions and how they are constructed in JavaScript (refer to the“Functions in JavaScript” section later in this chapter). For now, let’s start with a working definitionof a function as some set of instructions that performs an action or returns a value. Becausea function can return a value, it can return a Boolean true or false. Furthermore, youcan use a function call in an if statement as the test. Listing.14.16 shows how you might implementthis.

Listing. 14.16 Using a Function as a Conditional

```
if ( pageIsLoaded) {
alert (“All Done!”);
done = true;
}
else {
done = false; }
```

Loop Statements:

Sometimes you want to execute a series of statements over and over again until some conditionis met. An example of this is to play a sound in the background of your page until the userclicks Stop! or to repeatedly divide some number by 6 until it is less than 50. This action isperformed in JavaScript by the for and the while structures.for A for loop repeats some series of statements until some condition is met. The for loopstructure is virtually identical to the structure in Java. Its syntax is:

```
for ([some initial expression] ; [condition] ; [increment expression] ) {
some expressions...
}
```

You build a for loop by setting up three expressions that follow a more or less standard format.The initial expression can be of any degree of complexity, but it usually is simply an initialassignment of value to the counter variable. In the second expression, the condition is executedonce for each pass through the expressions. If the expression evaluates to true, thenthe block of expressions is executed. If the expression evaluates to false, the for loop iscompleted and the interpreter jumps down the next expression after the loop. The incrementexpression is evaluated after each pass through the loop and is usually where the “counter”variable is incremented or decremented. Essentially, this means you initialize some counter,test some condition, execute the enclosed statements if true, increment the counter, test thecondition again, and so on.Although not required, you should use the increment expression to change some value that willeventually render the condition expression false. Otherwise, your for loop will run forever (or ntil youget tired of waiting and reboot your computer).

Listing.14.17 gives you a simple example of a for loop in JavaScript.

Listing. 14.17 An Example of a for Loop

```
<script language=”JavaScript”>
var myMessage = “Here we go again! <br>”;
var numberOfRepeats = 100;
for ( i=0; i < numberOfRepeats ; i++) {
document.write(myMessage);
}
</script>
```

while The while loop is a simpler version of the for loop in that it tests some expressioneach time around and escapes if that expression is false. You will probably use while loopswhen the variable you are testing for is also present inside the statement block that you areexecuting during each loop. Note that the condition is tested first before the statements areexecuted, and that the condition is tested only once for each loop. Here is the standard syntaxfor a while loop:

14.32 CONTROL STATEMENTS

```
while (somecondition) {
some statements;
}
```

Listing. 14.18 repeatedly displays a series of lines that state the current value of tt until it isgreater than or equal to xx, which, in this case, is 55.

Listing. 14.18 An Example of a while Loop

```
<script language=”JavaScript”>
tt = 0
xx = 55
while ( tt <= 55) {
tt += 1;
document.write (“The value of tt is “ + tt +”. <br> “);
}
</script>
```

break and continue:

Sometimes you might want to have a finer degree of control over your block of statementswithin a for or while loop. Occasionally, you might want to arbitrarily jump out of a loop andcontinue down to the next statement. Or, you might want to stop the execution of statements inthe current loop and start a new loop. You can achieve both of these options by using breakand continue. break causes the for or while loop to terminate prematurely and the executionto jump down to the next line after the loop. continue stops the current loop and begins a newone.It is easy to get these statements confused, and their purpose may become unclear over time. An easyway to remember how these work is to think of break as breaking the loop, which renders the loopinoperable. Then, the program continues down. You can think of continue as a way of skipping whatever is below it and starting again. Listings 14.19 and 14.20 mirror Listings 14.17 and 14.18 andillustrate these control statements.

Listing. 14.19 Breaking Out of a for Loop

```
<script language=”JavaScript”>
var myMessage = “Here we go again! <br>”;
var numberOfRepeats = 100;
for ( i=0; i < numberOfRepeats ; i++) {
document.write(myMessage);
if ( i <0) {
document.write(“Invalid Number!”);
break
}
}
</script>
```

T I P

Listing 14.20 Continuing a while Loop

```
<script language=”JavaScript”>
var tt = 0;
xx = 55
while ( tt <= 55) {
tt += 1;
if (tt < 0) {
continue;
}
document.write (“The value of tt is “ + tt +”. <br> “);
}
</script>
```

Comments

Every language needs to have some way to document exactly what is going on, especially ifyou ever intend to reuse your code. It may seem obvious to you when you are deep in the zoneof programming your cool new script. But a few days later, you may find yourself wondering,“What was I thinking?” It’s always a good idea to comment your code. I talk about commentshere, in control statements, essentially because they are a way of telling the JavaScript interpreterto skip over some piece of code or comments, no matter what.Comments are similar to a for loop that is initially and always false. JavaScript supports twokinds of comments:

- Line-by-line version (//)
- Multiple-line version (/* ... */)

You can place anything you want in either of these comments, except for one thing. Do youremember when I talked about using HTML comments to keep the older browsers from erroneouslydisplaying JavaScript code? In other words, you cannot use --> in your commentsunless you are really intending the script to end.

Notice also that you must place the single-line comment in front of the HTML end commentnotation. This is because the JavaScript interpreter does not recognize --> as anything meaningfuland gives you an error if you forget to use // before it.Why, then, doesn’t the initial line (something such as “Hide me from old browsers”) after thebeginning HTML comment give you a JavaScript error? The reason is that the interpreterignores everything else on the line containing <--. This is handy for you, because you can usethis line to describe your script, and so forth. Listing.13 shows both ways of displaying comments.

Listing.14.21 An Example of Displaying Comments

```
<html>
<script language = “JavaScript”>
<!-- Hide this code from old browsers
one = 1
two = 2
// three = 99 everything on this line is ignored....
four = 4 ;
five = 5 ; /* everything on this line, and all
subsequent lines will be ignored, until
we get to the closing comment */
six = 6;
// remember to comment out the last line if you are using the HTML comments also
-->
```

```
//You must not have JavaScript if you see this line...
</script>
</html>
```

14.33 FUNCTIONS IN JAVASCRIPT

You have now reached one of the most interesting parts of JavaScript. The heart of most scriptsthat you build will consist of functions. You can think of a function as a named series of statementsthat can accept other variables or statements as arguments. Remember how the ifstatement was constructed:

```
if (someTest) {
zero or more statements
}
```

You build functions in a very similar way:

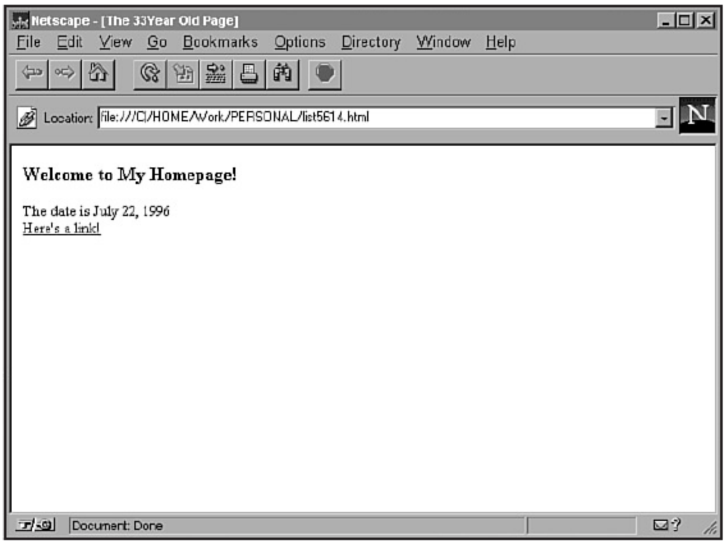
```
function someFunction (arguments) {
some statements
return someValue;
}
```

Let’s discuss functions in greater detail. As I mentioned earlier in this chapter, functions areblocks of code that you can reuse over and over again just by calling the blocks by name andoptionally passing some arguments to them. Functions form the heart of most of the scriptsyou will build and are almost as fundamental to JavaScript as classes are to Java.You will see that JavaScript comes with many built-in functions for you to use and allows you tocreate your own as well. Suppose, for instance, that you want to use JavaScript to create a smallHTML page. You can use functions to pull out each of the subtasks you want to do, whichmakes your code much easier to modify, read, and reuse. Let’s look at Listing.14.22.

Listing.14.22 A Simple Example Using Functions

```
<html>
<head>
<script language=”javaScript”>
<!-- remember me?
var age = 0;
function myHeader (age) {
document.write(“<TITLE>The “ + age + “Year Old Page</TITLE>”);
}
function myBody (date, color) {
document.write (“ <body bgcolor=” + color + “ >”);
document.write (“<h3>Welcome to My Homepage!</h3>”);
document.write (“The date is “ + date + “<br>”);
}
function manyLinks (index) {
if (index == 1) {
return “http://www.yahoo.com”;
}
else if (index == 2){
return “http://home.netscape.com”;
}
else return “http://www.idsoftware.com” ;
}
// return the title
myHeader(33);
// done for the moment! -->
</script>
</head>
<script language=JavaScript>
<!--
myBody(“July 22, 1996”, “#ffffff”);
document.write(“<a href=” + manyLinks(2) + “>Here’s a link!</a>”);
// -->
</script>
```

In this example, each function encapsulates some HTML code. You can see how you passinformation into each function by means of the arguments. JavaScript passes values by reference,meaning that when you pass a value to a function, you are really just passing a valuepointer to the function. (A value pointer is just an address, similar to how a house address onan envelope gives information about how to find the house.) If the function modifies that value,the value is changed for the entire script, not just the scope of the function. The result of thecode is shown in Figure 55.1.Also, notice the behavior of return. You can optionally return an explicit value back to the statementthat called the function (as in return http://...). The value returned can be of any validJavaScript type. If no value is explicitly returned, JavaScript returns true upon successfulcompletion of the function.Notice the difference between defining the function and calling the function. You define (orstore into memory) the function by using the function keyword. None of the statements insidethe function are executed until the function is called by using the function name elsewhere inthe script.


Netscape - [The 33Year Old Page]
File Edit View Go Bookmarks Options Directory Window Help
Document: Done

Location: file:///C|/HOME/Work/PERSONAL/list5614.html

Welcome to My Homepage!

The date is July 22, 1996

Here's a link!

14.34 ARRAYS

While I am on the subject of functions, it is convenient to introduce another extremely usefulconstruct in JavaScript—the array. An array is simply an ordered set of values that can beaccessed by a common name and an index (a number representing at what place in the seriesthat value is located). Before Netscape 3.0, you were forced to create arrays yourself by using afunction you will see quite often in scripts on the Internet. Listing.14.23 shows how to create afunction that builds an array for you.

FIG. 55.1

Output from Listing 14.23.

Listing.14.23 An Array Builder

```
function MakeArray(n) {
this.length = n;
for (var i = 1; i <= n; i++;) {
this[i] = “ “ }
return this
}
}
```

You may notice a new keyword here called this. this is a special keyword that refers to thecurrent object. I talk about this and another keyword you haven’t encountered, new, later in thissection. To create a new array, you simply assign the results of MakeArray to some name, asshown here:

```
Letterman = new MakeArray(10);
```

The new keyword is a way of telling JavaScript that the function to the right of it is an objectconstructor, and JavaScript treats it accordingly. To access values in your new array or set anyof the values, use this syntax:

Letterman[1] = “A list”

Letterman[3] = “Not so popular”

In Netscape 3.0, arrays are built in, so all you need to do is use Array instead of your MakeArrayfunction. In the previous case, this would be:

Letterman = new Array();

You can either set the size of the array when you initialize it or assign some null value to thehighest element in the array.

14.35 BUILT-IN FUNCTIONS

There are a few built-in functions in JavaScript. Table 14.4 lists them with a short description ofthe function of each.

Table 14.4 Built-In FunctionsFunction Descriptionescape(str) Converts strings to HTML special characters (such as " "

to %20).

unescape(str) Inverse of escape(). %20 to " ".

eval (str) Evaluates a string str as a JavaScript expression.

parseFloat (str, radix) Converts a string to a floating-point number (if possible).

parseInt (str) Converts a string to an integer value (if possible).

14.36 OBJECTS

Now that I have touched on functions that group together statements, let’s look at the equivalentstructure for data in general—the all-important object.

Because you surely have read some part of the rest of this book (unless you decided to skip tothis part first!), you have come face to face with Java objects. Basically, objects are a way oforganizing data and the manipulations you might associate with that data. In Java, you haveclasses and methods, but in JavaScript, you have objects and functions. As I mentioned before,JavaScript comes preloaded with many very useful objects and functions. This section familiarizesyou with Netscape’s object model and summarizes each of the many built-in objects.

Dot Notation :

JavaScript borrows from Java the system of accessing properties and methods (JavaScriptfreely mixes the terms function and method) by the use of the dot notation.Basically, you access information by first naming the top-level object that contains it, as well asall subsequent objects (or methods) that focus in on that information. Suppose you have anobject called car that contains an object called door. Suppose door contains another objectcalled doorhandle that uses a method called openDoor(). You could use this method at anytime by using this syntax:

car.door.doorhandle.openDoor()

Let’s say also that the door object has an attribute called color, and that color has a value of“Red”. You could assign that value to another variable by using a notation similar to this:myColor = car.door.color.value ;

Methods and Properties :

JavaScript objects contain data in the forms of properties and methods. Properties are basicallynamed values that are associated with a given object. Properties are accessed through thatobject. In the previous example of the car, door would have the property of color.Properties are handy and intuitive ways of storing information about an object. Methods (orfunctions) tend to be blocks of code that perform some operations on the object’s properties.Or, methods perhaps store their results in one of the properties. The openDoor() function is amethod of the object doorhandle. When I discuss the objects that are built in to Navigator, Icover their associated methods and properties as well.

The Window Object :

The Window object is the top-level object in JavaScript. It contains all other objects except thenavigator object, which is not tied to any particular window. Because most of your work is doneinside a Navigator window, this is a useful object that you should become familiar with.The Window object contains methods to open and close windows, bring up an alert dialog box(where you just click OK), bring up a confirm dialog box (you click Yes or No), and bring up aprompt dialog box (where you type in some information). The Window object also containsproperties for all frames that window contains and all child windows Window creates. It alsoallows you to change the status line at the bottom of the window (where you see those tickertapemessages on many pages).

Table 14.5 lists all of the properties and methods of the Window

Object Properties	**Description**
defaultStatus	The default message in the status bar.
document	The current document contained in the window.

frames	An array that describes all of the frames (if any) in the window..
length	Reflects the number of frames (if any) in the window.
name	The name of the window.
parent	Synonymous with the name of the window. Contains the frameset tags.
self	Synonymous with the name of the window and refers to the currentwindow.
status	Value appears in the window’s status bar. Usually only lasts a momentbefore overwritten by some other event.
top	Synonymous with the name of the window and represents the topmostwindow.
window	Synonymous with the name of the window and refers to the currentwindow.
location	A string specifying the URL of the current document.
alert	Brings up an alert dialog box.
close	Closes the window.
confirm	Brings up a dialog box with Yes or No buttons and a user-specifiedmessage.
open	Opens a new window.
prompt	Brings up a window with user-specified text and an input box thatallows the user to type in information.
setTimeout	Sets a time in milliseconds for an event to occur.
clearTimeout	Resets value set by setTimeout.

1321

The Document Object :

The Document object is extremely useful because it contains so much information about thecurrent document, and it can create HTML on-the-fly with its write and writeln methods.Table 14.6 lists the properties and methods of the document object, as well as short descriptionsof their purpose.

Table 14.6 Properties and Methods of the Document Object

Properties	**Description**
alinkColor	Reflects the ALINK attribute (in the <body> tag).
Anchors	An array listing all of the HTML anchors in the document (<a name>).

anchor	An anchor object.
bgColor	Reflects the value of the BGCOLOR attribute.
Cookie	Reflects the value of a Netscape cookie.
fgColor	The value of the TEXT attribute (in the <body> tag).
forms	An array listing all the forms in the document.
form	A form object.
history	An object containing the current browser history (links visited, number of links visited, and link URLs).
lastModified	The date the document was last modified.
linkColor	Reflects the LINK attribute of the <body> tag.
links	An array of all HTML links in the document (<a href>).
Link	A link object.
location	The URL of the document.
referrer	The URL of the document that called the current document.
title	Reflects the title of the document.
VlinkColor	Reflects the color listed in the VLINK attribute.
clear	Clears the window of all content.
close	After an open causes the string buffer to be written to the screen.
open	Begins a string to be written to the screen. Needs a close to actuallyforce the writing.
write	Writes some expression to the current window.
writeln	Same as write but adds a newline character at the end.
Objects	

The Form Object:

This object is created every time JavaScript encounters a <form>...</form> in your HTMLdocuments. It contains all of the information stored in your form and can be used to submitinformation to a function or back to the server. Table 14.7 describes the properties and methodsof the Form object.Table 14.7 Properties and Methods of the Form Object

Properties	**Description**
action	Reflects the HTML ACTION attribute of the <form> tag.
button	A button object (<input type=button>).
checkbox	A checkbox object (<input type= checkbox>).
elements	An array listing all elements in a form.

encoding	The value of the ENCTYPE attribute (for HTML uploads in Netscape).
hidden	A hidden object (<input type=hidden>).
Length	The number of elements in the form.
method	The METHOD attribute of <form>.
password	A password object (<input type=password>).
radio	A radio object (<input type=radio>).
reset	A reset button object.
select	A select object (<select>...<select>).
submit	A submit button object.
target	The TARGET attribute of <form>.
text	A text object (<input type=text>).
textarea	A textarea object (<textarea>...</textarea>).
submit	Submits the form to the location in the ACTION attribute.

The Navigator Object:

The Navigator object is distinct from the window object in that it contains information about thebrowser that persists across any given window. In Netscape 3.0, JavaScript adds two new properties—an object called mimeTypes, which lists all of the mimeTypes the browser can handle,and plug-ins, which lists all of the registered plug-ins the browser can use. Table 14.8 summarizesthe properties of the Navigator object (it has no associated methods).

Table 14.8 Navigator Object Properties

Properties	**Description**
appCode	Name The code name of the browser, such as “Mozilla.”
appName	The name of the browser, such as “Netscape.”
appVersion	Contains the version information of the browser, such as “2.0 (Win95,I).”
userAgent	Contains the user-agent header that the browser sends to the server toidentify itself, such as “Mozilla/2.0 (Win95, I).”
mimeTypes	An array reflecting all possible MIME types the browser can either handle itself or pass on to a plug-in or helper application(Netscape 3.0).
plug-ins	An array of registered plug-ins that the browser currently has loaded.

The String Object:

Other objects are built in to JavaScript that are not specific to either the browser or the window.The first of these is the String object. This object is very useful because you can use its methodsto modify and add HTML modifications without changing the string itself. One thing tonotice about this object is that you can string together any number of its methods to createmultilayers of HTML encoding. For example:

“Hello!”.bold().blink()

would return:

<blink><b>Hello!</b></blink>

Table 14.9 describes this object.

Table 14.9 Properties and Methods of the String Object

Property Description

length The number of characters in the string.

Methods Description

anchor Converts string to an HTML anchor.

big Encloses string in <big>...</big>.

blink Encloses string in <blink>...</blink>.

bold Encloses string in <b>...</b>.

charAt Returns the character at some index value. Index reads from left toright. If char not found, it returns a -1.

fixed Encloses string in <tt>...</tt>.

fontcolor Encloses string in <font color=somecolor>...</font>.

indexOf Looks for the first instance of some string and returns the index of thefirst character in the target string, or gives a -1 if not found.italics Encloses string in <i>...</i>.

lastIndexOf Same as indexOf, only begins searching from the right to find the lastinstance of the search string, or -1 if not found.link Converts string into a hyperlink.

small Encloses string in <small>...</small>.

strike Encloses string in <strike>...</strike>.

sub Encloses string in <sub>...</sub>.

substring Given a start and end index, returns the string contained by thoseindices.

sup Encloses string in <sup>...</sup>.

toLowerCase All uppercase characters are converted to lowercase (UpPeRcAsEbecomes uppercase).toUpperCase All lowercase characters are converted to uppercase.

The Math Object:

The Math object is both a set of methods that allows you to perform higher-level mathematicaloperations on your numerical data and a set of properties that contain some common mathematicalconstants. You can use the Math object anywhere in your scripts, as long as you referencethe methods like this:Math.PI

Or you can use the with keyword to contain a series of math statements:

with (Math) {

```
foo = PI
bar = sin(foo)
baz = tan(bar/foo)
}
```

Table 14.10 gives you a list of the Math properties and methods.

Table 14.10 Math Properties and Methods

Properties	Methods
E	abs
LOG2E	acos
SQRT1_2	asin
LN2	atan
LOG10E	ceil
SQRT2	cos
LN10	exp
PI	floor
log	
max	
min	
pow	
random	
round	
sin	
sqrt	
tan	

The Date Object :

The final object I examine here is the Date object. This object allows you to grab informationabout the client's current time, year, month, date, and more. In addition, you can quickly createnew date objects that can simplify keeping track of dates or time intervals between events. Youcan even parse a text string for date information that can be used elsewhere as a Date object.This object is most commonly used to create dynamic clocks, change page attributes (such asthe background color) based on the time of day, and so on. Table 14.11 gives you a view of theDate object's methods (it has no properties).

Table 14.11 Methods of the Date Object

Method	**Description**
getDate	Returns the current date.
getDay	Returns the day of the week from a date object.
getHours	Returns the current number of hours since midnight.

getMinutes	Returns the current number of minutes past the hour.
getMonth	Returns the number of months since January.
getSeconds	Returns the number of seconds past the minute.
getTime	Returns the current time from the specified date object.
getTimeZoneOffset	Returns the offset in minutes for the current location (eitherore or less than GMT, or Greenwich Mean Time).
getYear	Returns the year from the Date object.
parse	Returns the number of milliseconds since January 1, 1970 00:00:00 for the current locale from the Date object.
setDate	Argument used to set a Date object.
SetHours	Argument sets the hours of the Date.
setMinutes	Argument sets the minutes of the Date.
setMonth	Argument sets the month value.
setSeconds	Argument sets the seconds value.
setTime	Argument sets the time value of the specified Date object.
setYear	Argument sets the year value for the specified Date object.
	toGMTString Converts a date to a string using the standard GMT conventions(for example, Wed, 24 Jul 12:49:08 GMT).
toLocaleString	Converts a date to a string but is aware of the locale's convention instead of GMT. (7/24/96 10:50:02).

UTC Opposite of toGMTString. Converts a string into the number ofmilliseconds since the epoch.).

14.37 A FINAL EXAMPLE

As a final example of what JavaScript can do, Listing.16 is the source code for a Web pagethat displays the current time every second. You can see all of the elements that have beendiscussed previously in this chapter somewhere within this example. Essentially, this programgets a Date object every second; parses that object for the current minutes, seconds, andhours; converts those values to a string; and then sets a form input field to that value. Using aform in this way is quite common in JavaScript. Instead of being a way to input data, the textinput field becomes a “screen” to display the time.

Listing.14.24 A JavaScript Clock

```
<HTML>
<HEAD>
<TITLE>JavaScript Clock</TITLE>
<script Language=”JavaScript”>
<!-- Hide me from old browsers - hopefully
// Netscapes Clock - Start
// this code was taken from Netscapes JavaScript documentation at
// www.netscape.com on Jan.25.96
var timerID = null;
var timerRunning = false;
function stopclock (){
if(timerRunning)
clearTimeout(timerID);
timerRunning = false;
}
function startclock () {
// Make sure the clock is stopped
stopclock();
showtime();
}
function showtime () {
var now = new Date();
var hours = now.getHours();
var minutes = now.getMinutes();
var seconds = now.getSeconds()
var timeValue = “” + ((hours >12) ? hours -12 :hours)
timeValue += ((minutes < 10) ? “:0” : “:”) + minutes
timeValue += ((seconds < 10) ? “:0” : “:”) + seconds
timeValue += (hours >= 12) ? “ P.M.” : “ A.M.”
document.clock.face.value = timeValue;
// you could replace the above with this
// and have a clock on the status bar:
// window.status = timeValue;
timerID = setTimeout(“showtime()”,1000);
timerRunning = true;
}
// Netscapes Clock - Stop
// end -->
</script>
</HEAD>
<BODY bgcolor=”#ffffff” text=”#000000" link=”#0000ff”
alink=”#008000" vlink=”800080" onLoad=”startclock()”>
<!-- main -->
```

```
<table >
<tr>
<td colspan=3>
<form name=”clock” onSubmit=”0">
```

continues

A Final Example

```
<div align=right>
<input type=”text” name=”face” size=12 value=””>
</div>
<center><b><font size=-1 >Welcome to My HomePage!</font></b></center><p>
</table>
</BODY>
</HTML>
```

Let’s go through this script and see how it works to create the changing clock you will see inyour browser.After the initial HTML code starting the page, the browser sees the <SCRIPT> tag and begins topass the code into the JavaScript interpreter. The HTML comment <-- hides the JavaScriptcode from old browsers.

The next three lines are comments that JavaScript ignores.

The next two lines initialize timerID to null (a special value that acts as a placeholder) andtimerRunning to false (a Boolean value). The variable timerID is used in the setTimeOut andclearTimeOut function. It just acts as a name to keep track of that specific countdown.The next five lines define a function called stopclock which tests if the timerRunning value istrue. If so, it calls clearTimeout which frees up the countdown timer called timerID.The next five lines (after a space) define a function called startclock. All startclock does iscall stopclock and then the function showtime. It’s important to stop the clock before callingshowtime, because showtime resets the countdown timer timerID.The next 16 lines define the heart of the script, called showtime. This function creates a newDate object called now and gets the hours, minutes, and seconds values from that object andassigns them to the variables hour, minutes, and seconds, respectively. By creating this newobject every time showtime is called, the script is getting the most recent time possible, whichis why the clock changes every second.

After the hours, minutes, and seconds are retrieved from the Date object, a new variabletimeValue is created, which is a String object, and it assigns the corrected value of hours tothis string. (The (hours >12) ? hours -12 :hours expression converts the hours from 24-hour time to 12-hour time.) The next timeValue assignments append the values of minutes andseconds to the timeValue string—correcting for tens of minutes.

The linedocument.clock.face.value = timeValueplaces the resulting string into the form text input field that is defined later. By assigning thisvalue to that field, it causes that value to appear in that box on the page.The next line in the function showtime (following the three comments) starts a countdown ofone second and calls it timerID. After one second, the function showtime is called again—essentially, this is a way of calling a this function over and over again every second.

The last line in the showtime function sets the timerRunning value to the Boolean true whichwould affect the stopclock function (breaking the one-second loop which timerID had beencausing). To test this, run this script and then in the URL input window (at the top of thebrowser window), type:
javascript:stopclock()

You see that the clock stops. Typing

javascript:startclock()

in the URL gets the clock running again.

After the function showtime, the rest of the lines close out the script and create via HTML atable that contains a form called clock with one input field called face.Notice that, in the <BODY> tag the onLoad="startclock()" statement, after the entire page isloaded into the window, the onLoad event handler is triggered, and the startclock function iscalled, which begins the script.

14.38 SUMMARY

This chapter covers what are data structures? Collections, *Vector, Hashtable, Properties, Stack,Date,BitSet, StringTokenizer, Random, Observable.*

This chapter also covers Java and JavaScript, features of javascript, JavaScript and Java Integration, Basics , Events, Event Handlers, Variables, Variable Names , Variable Scope, Literals, Expressions and Operators, Control Statements, Functions in JavaScript, Arrays, Built-In Functions, Objects

14.39 QUESTION

1. What are data structures?
2. What are the features of Java Script.?
3. Differentiate between java and JavaScript.
4. What are the different objects of JavaScript?
5. What are the inbuilt functions of Java Script.

*****

15

SERVLETS

Unit Structure

15.1 BACKGROUND

In order to understand the advantages of servlets, you must have a basic understandingof how Web browsers and servers cooperate to provide content to a user. Considera request for a static Web page. A user enters a Uniform Resource Locator (URL) intoa browser. The browser generates an HTTP request to the appropriate Web server.The Web server maps this request to a specific file. That file is returned in an HTTPresponse to the browser. The HTTP header in the response indicates the type of thecontent. The Multipurpose Internet Mail Extensions (MIME) are used for this purpose.For example, ordinary ASCII text has a MIME type of text/plain. The HypertextMarkup Language (HTML) source code of a Web page has a MIME type of text/html.

Now consider dynamic content. Assume that an online store uses a database tostore information about its business. This would include items for sale, prices, availability,orders, and so forth. It wishes to make this information accessible to customers viaWeb pages. The contents of those Web pages must be dynamically generated in orderto reflect the latest information in the database.In the early days of the Web, a server could dynamically construct a page by creatinga separate process to handle each client request. The process would open connectionsto one or more databases in order to obtain the necessary information. It communicatedwith the Web server via an interface known as the Common Gateway Interface (CGI).CGI allowed the separate process to read data from the HTTP request and write data tothe HTTP response. A variety of different languages were used to build CGI programs.These included C, C++, and Perl.However, CGI suffered serious performance problems. It was expensive in termsof processor and memory resources to create a separate process for each client request.It was also expensive to open and close database connections for each client request.In addition, the CGI programs were not platform-independent. Therefore, othertechniques were introduced. Among these are servlets.Servlets offer several advantages in comparison with CGI. First, performance issignificantly better. Servlets execute within the address space of a Web server. It isnot necessary to create a separate process to handle each client request. Second, servletsare platform-independent because they are written in Java. A number of Web serversfrom different vendors offer the Servlet API. Programs developed for this API can bemoved to any of these environments without recompilation. Third, the Java securitymanager on the server enforces a set of restrictions to protect the resources on a server950 Java ™ 2 : The Complete Referencemachine. You will see that some servlets are trusted and others are untrusted. Finally,the full functionality of the Java class libraries is available to a servlet. It can communicatewith applets, databases, or other software via the sockets and RMI mechanisms thatyou have seen already.

15.2 THE LIFE CYCLE OF A SERVLET

Three methods are central to the life cycle of a servlet. These are **init( )**, **service( )**,and **destroy( )**. They are implemented by every servlet and are invoked at specifictimes by the server. Let us consider a typical user scenario to understand when thesemethods are called.

First, assume that a user enters a Uniform Resource Locator (URL) to a Webbrowser. The browser then generates an HTTP request for this URL. This request isthen sent to the appropriate server.Second, this HTTP request is received by the Web server. The server maps thisrequest to a particular servlet. The servlet is dynamically retrieved and loaded intothe address space of the server.Third, the server invokes the **init( )** method of the servlet. This method is invokedonly when the servlet is first loaded into memory. It is possibleto pass initializationparameters to the servlet so it may configure itself.Fourth, the server invokes the **service( )** method of the servlet. This method iscalled to process the HTTP request. You will see that it is possible for the servlet toread data that has been provided in the HTTP request. It may also formulate anHTTP response for the client.

The servlet remains in the server’s address space and is available to process anyother HTTP requests received from clients. The **service( )** method is called for eachHTTP request.

Finally, the server may decide to unload the servlet from its memory. The algorithmsby which this determination is made are specific to each server. The server calls the**destroy( )** method to relinquish any resources such as file handles that are allocated forthe servlet. Important data may be saved to a persistent store. The memory allocatedfor the servlet and its objects can then be garbage collected.

Using Tomcat For Servlet Development:

To create servlets, you will need to download a servlet development environment. Theone currently recommended by Sun is Tomcat 4.0, which supports the latest servletspecification, which is 2.3. (The complete servlet specification is available for downloadthrough **java.sun.com**.) Tomcat replaces the old JSDK (Java Servlet Development Kit)that was previously provided by Sun. Tomcat is an open-source product maintainedby the Jakarta Project of the Apache Software Foundation. It contains the class libraries,documentation, and run-time support that you will need to create and test servlets.

You can download Tomcat through the Sun Microsystems Web site at **java.sun.com**.The current version is 4.0. Follow the instructions to install this toolkit on yourmachine. The examples in this chapter assume a Windows environment. The defaultlocation for Tomcat 4.0 isC:\Program Files\Apache Tomcat 4.0\This is the location assumed by the examples in this book. If you load Tomcat in adifferent location, you will need to make appropriate changes to the examples. You mayneed to set the environmental variable **JAVA_HOME** to the top-level directory in whichthe Java Software Development Kit is installed. For Java 2, version 1.4, the defaultdirectory is **C:\j2sdk1.4.0**, but you will need to confirm this for your environment.To start Tomcat, select Start Tomcat in the Start | Programs menu, or run **startup.bat**from theC:\Program Files\Apache Tomcat 4.0\bin\directory. When you are done testing servlets, you can stop Tomcat by selecting StopTomcat in the Start | Programs menu, or run **shutdown.bat**.

The directoryC:\Program Files\Apache Tomcat 4.0\common\lib\contains **servlet.jar**. This JAR file contains the classes and interfaces that are neededto build servlets. To make this file accessible, update your **CLASSPATH** environmentvariable so that it includesC:\Program Files\Apache Tomcat 4.0\common\lib\servlet.jar.Alternatively, you can specify this class file when you compile the servlets. Forexample, the following command compiles the first servlet example:javac HelloServlet.java -classpath "C:\Program Files\Apache Tomcat

4.0\common\lib\servlet.jar"Once you have compiled a servlet, you must copy the class file into the directorythat Tomcat uses

for example servlet class files. For the purposes of this chapter, youmust put the servlet files into the following directory:
C:\Program Files\Apache Tomcat 4.0\webapps\examples\WEB-INF\classes

15.3 A SIMPLE SERVLET

To become familiar with the key servlet concepts, we will begin by building and testinga simple servlet. The basic steps are the following:

1. Create and compile the servlet source code.
2. Start Tomcat.
3. Start a Web browser and request the servlet.

Let us examine each of these steps in detail.

Create and Compile the Servlet Source Code :

To begin, create a file named **HelloServlet.java** that contains the following program:

Listing 15.1

```
import java.io.*;
import javax.servlet.*;
public class HelloServlet extends GenericServlet {
public void service(ServletRequest request,
ServletResponse response)
throws ServletException, IOException {
response.setContentType("text/html");
PrintWriter pw = response.getWriter();
pw.println("<B>Hello!");
pw.close();
}
}
```

Let’s look closely at this program. First, note that it imports the **javax.servlet** package.This package contains the classes and interfaces required to build servlets. You will learnmore about these later in this chapter. Next, the program defines **HelloServlet** as a subclassof **GenericServlet**. The **GenericServlet** class provides functionality that makes it easy tohandle requests and responses.Inside **HelloServet**, the **service( )** method (which is inherited from **GenericServlet**)is overridden. This method handles requests from a client. Notice that the first argumentis a **ServletRequest** object. This enables the servlet to read data that is provided viathe client request. The second argument is a **ServletResponse** object. This enables theservlet to formulate a response for the client.The call to **setContentType( )** establishes the MIME type of the HTTP response.In this program, the MIME type is text/html. This indicates that the browser shouldinterpret the content as HTML source code.

Next, the **getWriter( )** method obtains a **PrintWriter**. Anything written to this stream is sent to the client as part of the HTTP response. Then **println( )** is used to write some simple HTML source code as the HTTP response. Compile this source code and place the **HelloServlet.class** file in the Tomcat class files directory as described in the previous section.

Start Tomcat :

As explained, to start Tomcat, select Start Tomcat in the Start **|** Programs menu, or run**startup.bat** from theC:\Program Files\Apache Tomcat 4.0\bin\directory.

Start a Web Browser and Request the Servlet

Start a Web browser and enter the URL shown here:

http://localhost:8080/examples/servlet/HelloServlet

Alternatively, you may enter the URL shown here:

http://127.0.0.1:8080/examples/servlet/HelloServlet

This can be done because 127.0.0.1 is defined as the IP address of the local machine. You will observe the output of the servlet in the browser display area. It will contain the string **Hello!** in bold type.

15.4 THE SERVLET API

Two packages contain the classes and interfaces that are required to build servlets. These are **javax.servlet** and **javax.servlet.http**. They constitute the Servlet API. Keep in mind that these packages are not part of the Java core packages. Instead, they are standard extensions. Therefore, they are not included in the Java Software Development Kit. You must download Tomcat to obtain their functionality. The Servlet API has been in a process of ongoing development and enhancement. The current servlet specification is version is 2.3 and that is the one used in this book. However, because changes happen fast in the world of Java, you will want to check forany additions or alterations. This chapter discusses the core of the Servlet API, which will be available to most readers. The Servlet API is supported by most Web servers, such as those from Sun, Microsoft, and others. Check at **http://java.sun.com** for the latest information.The javax.servlet Package The **javax.servlet** package contains a number of interfaces and classes that establish the framework in which servlets operate. The following table summarizes the core interfaces that are provided in this package. The most significant of these is **Servlet**. All servlets must implement this interface or extend a class that implements the interface. The **ServletRequest** and **ServletResponse** interfaces are also very important. Interface DescriptionServlet Declares life cycle methods for a servlet.ServletConfig Allows servlets to get initialization parameters.ServletContext Enables servlets to log events and access informationabout their environment.

ServletRequest Used to read data from a client request.

ServletResponse Used to write data to a client response.

SingleThreadModel Indicates that the servlet is thread safe.

The following table summarizes the core classes that are provided in the**javax.servlet** package.

Class Description :

GenericServlet Implements the **Servlet** and **ServletConfig**interfaces.

ServletInputStream Provides an input stream for reading requests from a client.

ServletOutputStream Provides an output stream for writing responses to a client.

ServletException Indicates a servlet error occurred.

UnavailableException Indicates a servlet is unavailable.

Let us examine these interfaces and classes in more detail.

The Servlet Interface

All servlets must implement the **Servlet** interface. It declares the **init( )**, **service( )**, and **destroy( )** methods that are called by the server during the life cycle of a servlet. A method is also provided that allows a servlet to obtain any initialization parameters. The methods defined by **Servlet** are shown in Table-15.1.

The **init( )**, **service( )**, and **destroy( )** methods are the life cycle methods of the servlet. These are invoked by the server. The **getServletConfig( )** method is called bythe servlet to obtain initialization parameters. A servlet developer overrides the**getServletInfo( )** method to provide a string with useful information (for example,author, version, date, copyright). This method is also invoked by the server.

The ServletConfig Interface :

The **ServletConfig** interface is implemented by the server. It allows a servlet to obtainconfiguration data when it is loaded. The methods declared by this interface aresummarized here:

Method	**Description**
ServletContext getServletContext( )	Returns the context for this servlet.

String getInitParameter (String *param*)	Returns the value of the initializationparameter named *param*.
Enumeration	getInitParameterNames( ) Returns an enumeration of allinitialization parameter names.
String getServletName( )	Returns the name of the invoking servlet.
Method Description	void destroy( ) Called when the servlet is unloaded.
ServletConfig getServletConfig( )	Returns a **ServletConfig** object that contains
any initialization parameters.	
String getServletInfo( )	Returns a string describing the servlet.
void init(ServletConfig *sc*) throws ServletException	Called when the servlet is initialized.Initialization parameters for the servlet can beobtained from *sc*. An **UnavailableException**should be thrown if the servlet cannot beinitialized.
void service(ServletRequest *req*, ServletResponse *res*) throws ServletException,	IOException Called to process a request from a client. Therequest from the client can be read from *req*.The response to the client can be written to*res*. An exception is generated if a servlet orIO problem occurs.

The ServletContext Interface:

The **ServletContext** interface is implemented by the server. It enables servlets to obtaininformation about their environment. Several of its methods are summarized in Table-15.2.

The **ServletRequest** interface is implemented by the server. It enables a servlet toobtain information about a client request. Several of its methods are summarized in Table-15.3.

The **ServletResponse** interface is implemented by the server. It enables a servlet toformulate a response for a client. Several of its methods are summarized in Table-15.4.

The SingleThreadModel Interface:

This interface is used to indicate that only a single thread will execute the **service( )**method of a servlet at a given time. It defines no constants and declares no methods.If a servlet implements this interface, the server has two options. First, it can createseveral instances of the servlet. When a client request arrives, it is sent to an availableinstance of the servlet. Second, it can synchronize access to the servlet.

Method	**Description**
Object getAttribute(String *attr*)	Returns the value of the server attribute named *attr*.
String getMimeType(String *file*)	Returns the MIME type of *file*.
String getRealPath(String *vpath*)	Returns the real path that correspondsto the virtual path *vpath*.
String getServerInfo( )	Returns information about the server.
void log(String *s*)	Writes *s* to the servlet log.
void log(String *s*, Throwable *e*)	Write *s* and the stack trace for *e* to theservlet log.
void setAttribute(String *attr*, Object *val*)	Sets the attribute specified by *attr* to thevalue passed in *val*.

Table-15.2. Various Methods Defined by ServletContext

Method	**Description**
Object getAttribute(String *attr*)	Returns the value of the attribute named *attr*.
String getCharacterEncoding( )	Returns the character encoding ofthe request.
int getContentLength( )	Returns the size of the request. Thevalue –1 is returned if the size isunavailable.
String getContentType( )	Returns the type of the request. A**null** value is returned if the typecannot be determined.
ServletInputStream getInputStream( ) throws IOException	Returns a **ServletInputStream**that can be used to read binarydata from the

	request. An**IllegalStateException** is thrownif **getReader( )** has already beeninvoked for this request.
String getParameter(String *pname*)	Returns the value of the parameter named *pname*.
Enumeration getParameterNames( )	Returns an enumeration of the parameter names for this request.
String[ ] getParameter Values(String *name*)	Returns an array containing valuesassociated with the parameterspecified by *name*.
String getProtocol( )	Returns a description of Theprotocol.
BufferedReader getReader( ) throws IOException	Returns a buffered reader thatcan be used to read text from therequest. An **IllegalStateException**is thrown if **getInputStream( )** hasalready been invoked for thisrequest.

Table-15.3. Various Methods Defined by ServletRequest

Method	**Description**
String getRemoteAddr( )	Returns the string equivalent of theclient IP address.
String getRemoteHost( )	Returns the string equivalent of theclient host name.
String getScheme( )	Returns the transmission scheme ofthe URL used for the request (forexample, “http”, “ftp”).
String getServerName( )	Returns the name of the server.
int getServerPort( )	Returns the port number.

Table-15.3. Various Methods Defined by ServletRequest (continued)

Method	**Description**

String getCharacterEncoding( )	Returns the character encoding for theresponse.
ServletOutputStream getOutputStream( ) throws IOException	Returns a **ServletOutputStream** that can beused to write binary data to the response.An **IllegalStateException** is thrown if**getWriter( )** has already been invoked forthis request.
PrintWriter getWriter( ) throws IOException	Returns a **PrintWriter** that can be usedto write character data to the response.An **IllegalStateException** is thrown if**getOutputStream( )** has already beeninvoked for this request.
void setContentLength(int *size*)	Sets the content length for the response to *size*.
void setContentType(String *type*)	Sets the content type for the response to *type*.

Table-15.4. Various Methods Defined by ServletResponse

The GenericServlet Class:

The **GenericServlet** class provides implementations of the basic life cycle methods fora servlet and is typically subclassed by servlet developers. **GenericServlet** implementsthe **Servlet** and **ServletConfig** interfaces. In addition, a method to append a string tothe server log file is available. The signatures of this method are shown here:

void log(String *s*)
void log(String *s*, Throwable *e*)

Here, *s* is the string to be appended to the log, and *e* is an exception that occurred.The**ServletInputStream Class.** The **ServletInputStream** class extends **InputStream**. It is implemented by the serverand provides an input stream that a servlet developer can use to read the data from aclient request. It defines the default constructor. In addition, a method is provided toread bytes from the stream. Its signature is shown here:

int readLine(byte[ ] *buffer*, int *offset*, int *size*) throws IOException

Here, *buffer* is the array into which *size* bytes are placed starting at *offset*. The methodreturns the actual number of bytes read or –1 if an end-of-stream condition is encountered.

ServletOutputStream Class:

The **ServletOutputStream** class extends **OutputStream**. It is implemented by theserver and provides an output stream that a servlet developer can use to write datato a client response. A default constructor is defined. It also defines the **print( )** and**println( )** methods, which output data to the stream.

The Servlet Exception Classes :

javax.servlet defines two exceptions. The first is **ServletException**, which indicates thata servlet problem has occurred. The second is **UnavailableException**, which extends**ServletException**. It indicates that a servlet is unavailable.

15.5 READING SERVLET PARAMETERS

The **ServletRequest** class includes methods that allow you to read the names andvalues of parameters that are included in a client request. We will develop a servletthat illustrates their use. The example contains two files. A Web page is defined in**PostParameters.htm** and a servlet is defined in **PostParametersServlet.java**.The HTML source code for **PostParameters.htm** is shown in the following listing. Itdefines a table that contains two labels and two text fields. One of the labels is Employeeand the other is Phone. There is also a submit button. Notice that the action parameterof the form tag specifies a URL. The URL identifies the servlet to process the HTTPPOST request.

Listing 15.2

```
<html>
<body>
<center>
<form name="Form1"
method="post"
action="http://localhost:8080/examples/servlet/PostParametersS
ervlet">
<table>
<tr>
<td><B>Employee</td>
<td><input type=textbox name="e" size="25" value=""></td>
</tr>
<tr>
<td><B>Phone</td>
<td><input type=textbox name="p" size="25" value=""></td>
```

```
</tr>
</table>
<input type=submit value="Submit">
</body>
</html>
```

The source code for **PostParametersServlet.java** is shown in the following listing.The **service( )** method is overridden to process client requests. The **getParameterNames( )**method returns an enumeration of the parameter names. These are processed in a loop.You can see that the parameter name and value are output to the client. The parametervalue is obtained via the **getParameter( )** method.

Listing 15.3

```
import java.io.*;
import java.util.*;
import javax.servlet.*;
public class PostParametersServlet
extends GenericServlet {
public void service(ServletRequest request,
ServletResponse response)
throws ServletException, IOException {
// Get print writer.
PrintWriter pw = response.getWriter();


// Get enumeration of parameter names.
Enumeration e = request.getParameterNames();
// Display parameter names and values.
while(e.hasMoreElements()) {
String pname = (String)e.nextElement();
pw.print(pname + " = ");
String pvalue = request.getParameter(pname);
pw.println(pvalue);
}
pw.close();
}
}
```

Compile the servlet and perform these steps to test this example:

1. Start Tomcat (if it is not already running).
2. Display the Web page in a browser.

3. Enter an employee name and phone number in the text fields.
4. Submit the Web page.

After following these steps, the browser will display a response that is dynamicallygenerated by the servlet.

The javax.servlet.http Package

The **javax.servlet.http** package contains a number of interfaces and classes that arecommonly used by servlet developers. You will see that its functionality makes it easyto build servlets that work with HTTP requests and responses.The following table summarizes the core interfaces that are provided in this package:

Interface Description

HttpServletRequest Enables servlets to read data from an HTTP request.

HttpServletResponse Enables servlets to write data to an HTTP response.

HttpSession Allows session data to be read and written.

HttpSessionBindingListener Informs an object that it is bound to or unboundfrom a session.

The following table summarizes the core classes that are provided in this package.

The most important of these is **HttpServlet**. Servlet developers typically extend thisclass in order to process HTTP requests.

Class Description :

Cookie Allows state information to be stored on a clientmachine.

HttpServlet Provides methods to handle HTTP requests andresponses.

HttpSessionEvent Encapsulates a session-changed event.

HttpSessionBindingEvent Indicates when a listener is bound to or unboundfrom a session value, or that a session attributechanged.

The HttpServletRequest Interface

The **HttpServletRequest** interface is implemented by the server. It enables a servlet toobtain information about a client request. Several of its methods are shown in Table-15.5.

Method `	Description
String getAuthType( )	Returns authentication scheme.

Cookie[ ] getCookies( )	Returns an array of the cookies in this request.
long getDateHeader(String *field*)	Returns the value of the date header field named *field*.
String getHeader(String *field*)	Returns the value of the header field named *field*.
Enumeration getHeaderNames( )	Returns an enumeration of the headernames.
int getIntHeader(String *field*)	Returns the **int** equivalent of the header field named *field*.
String getMethod( )	Returns the HTTP method for thisrequest.
String getPathInfo( )	Returns any path information that islocated after the servlet path and beforea query string of the URL.
String getPathTranslated( )	Returns any path information thatis located after the servlet path andbefore a query string of the URL after translating it to a real path.
String getQueryString( )	Returns any query string in the URL.
String getRemoteUser( )	Returns the name of the user whoissued this request.
String getRequestedSessionId( )	Returns the ID of the session.
String getRequestURI( )	Returns the URI.
StringBuffer getRequestURL( )	Returns the URL.
String getServletPath( )	Returns that part of the URL thatidentifies the servlet.
HttpSession getSession( )	Returns the session for this request. If a session does not exist, one is Createdand then returned.
HttpSession getSession(boolean *new*)	If*new* is **true** and no session exists, creates and returns a session for this request. Otherwise, returns the existing session for this request.
boolean isRequestedSessionIdFromCookie( )	Returns **true** if a cookie contains the session ID. Otherwise, returns **false**.
Boolean isRequestedSessionId	

FromURL( )	Returns **true** if the URL contains the session ID. Otherwise, returns **false**.
boolean isRequestedSessionIdValid( )	Returns **true** if the requested session ID is valid in the current session context.

The HttpServletResponse Interface :

The **HttpServletResponse** interface is implemented by the server. It enables a servletto formulate an HTTP response to a client. Several constants are defined. Thesecorrespond to the different status codes that can be assigned to an HTTP response. Forexample, **SC_OK** indicates that the HTTP request succeeded and **SC_NOT_FOUND**indicates that the requested resource is not available. Several methods of this interfaceare summarized in Table-15.6.

Method	Description
void addCookie(Cookie *cookie*)	Adds *cookie* to the HTTP response.
boolean containsHeader(String *field*)	Returns **true** if the HTTP response header contains a field named *field*.
String encodeURL(String *url*)	Determines if the session ID must be encoded in the URL identified as *url*. If so, returns the modified version of *url*. Otherwise, returns *url*. All URLs generated by a servlet should be processed by this method.
String encodeRedirectURL(String *url*)	Determines if the session ID must be encoded in the URL identified as *url*. If so, returns the modified version of *url*. Otherwise, returns *url*. All URLs passed to **sendRedirect( )** should be processed by this method.
void sendError(int *c*) throws IOException	Sends the error code *c* to the client.
void sendError(int *c*, String *s*) throws IOException	Sends the error code *c* and message *s* to the client.
void sendRedirect(String *url*) throws IOException	Redirects the client to *url*.

Table-15.6. Various Methods Defined by HttpServletResponse

The HttpSession Interface

The **HttpSession** interface is implemented by the server. It enables a servlet to read andwrite the state information that is associated with an HTTP session. Several of its methodsare summarized in Table-15.7. All of these methods throw an **IllegalStateException** if thesession has already been invalidated.

Method	**Description**
void setDateHeader(String *field*, long *msec*)	Adds *field* to the header with date value equal to *msec* (milliseconds since midnight, January 1, 1970, GMT).
void setHeader (String *field*, String *value*)	Adds *field* to the header with valueequal to *value*.
void setIntHeader (String *field*, int *value*)	Adds *field* to the header with valueequal to *value*.
void setStatus(int *code*)	Sets the status code for thisresponse to *code*.
Object getAttribute(String *attr*)	Returns the value associated with thename passed in *attr*. Returns **null** if*attr* is not found.
Enumeration getAttributeNames( )	Returns an enumeration of the attributenames associated with the session.
long getCreationTime( )	Returns the time (in milliseconds sincemidnight, January 1, 1970, GMT) whenthis session was created.
String getId( )	Returns the session ID.

Table-15.7. The Methods Defined by HttpSession

The HttpSessionBindingListener Interface

The **HttpSessionBindingListener** interface is implemented by objects that need to benotified when they are bound to or unbound from an HTTP session. The methods thatare invoked when an object is bound or unbound are:

void valueBound(HttpSessionBindingEvent *e*)
void valueUnbound(HttpSessionBindingEvent *e*)

Here, *e* is the event object that describes the binding.

15.6 THE COOKIE CLASS

The **Cookie** class encapsulates a cookie. A cookie is stored on a client and contains stateinformation. Cookies are valuable for tracking user activities. For example, assume thata user visits an online store. A cookie can save the user’s name, address, and otherinformation. The user does not need to enter this data each time he or she visits the store.A servlet can write a cookie to a user’s machine via the **addCookie( )** method of the**HttpServletResponse** interface. The data for that cookie is then included in the headerof the HTTP response that is sent to the browser.

Method	Description
long getLastAccessedTime( )	Returns the time (in milliseconds sincemidnight, January 1, 1970, GMT) whenthe client last made a request for thissession.
void invalidate( )	Invalidates this session and removes itfrom the context.
boolean isNew( )	Returns **true** if the server created thesession and it has not yet beenaccessed by the client.
void removeAttribute(String *attr*)	Removes the attribute specified by *attr*from the session.
void setAttribute(String *attr*, Object *val*)	Associates the value passed in *val* withthe attribute name passed in *attr*.

Table-15.7. The Methods Defined by HttpSession (continued)

The names and values of cookies are stored on the user’s machine. Some of theinformation that is saved for each cookie includes the following:

- The name of the cookie
- The value of the cookie
- The expiration date of the cookie
- The domain and path of the cookie

The expiration date determines when this cookie is deleted from the user’s machine.If an expiration date is not explicitly assigned to a cookie, it is deleted when the currentbrowser session ends. Otherwise, the cookie is saved in a file on the user’s machine.The domain and path of the cookie determine when it is included in the header ofan HTTP request. If the user enters a URL whose domain and path match these values,the cookie is then supplied to the Web server. Otherwise, it is not.There is one constructor for **Cookie**. It has the signature shown here:
Cookie(String *name*, String *value*)Here, the name and value of the cookie are supplied as arguments to the constructor.The methods of the **Cookie** class are summarized in Table-15.8.

Method	Description
Object clone( )	Returns a copy of this object.

String getComment( )	Returns the comment.
String getDomain( )	Returns the domain.
int getMaxAge( )	Returns the age (in seconds).
String getName( )	Returns the name.
String getPath( )	Returns the path.
boolean getSecure( )	Returns **true** if the cookie must be sent usingonly a secure protocol. Otherwise, returns **false**.
String getValue( )	Returns the value.
int getVersion( )	Returns the cookie protocol version. (Will be0 or 1.)

Table-15.8. The Methods Defined by Cookie

The HttpServlet Class

The **HttpServlet** class extends **GenericServlet**. It is commonly used when developing servlets that receive and process HTTP requests. The methods of the **HttpServlet** class are summarized in Table-15.9.

Method	**Description**
void setComment(String *c*)	Sets the comment to *c*.
void setDomain(String *d*)	Sets the domain to *d*.
void setMaxAge(int *secs*)	Sets the maximum age of the cookie to *secs*. This is the number of seconds after which the cookie is deleted. Passing –1 causes the cookie to be removed when the browser is terminated.
void setPath(String *p*)	Sets the path to *p*.
void setSecure(boolean *secure*)	Sets the security flag to *secure*, which means that cookies will be sent only when a secure protocol is being used.
void setValue(String *v*)	Sets the value to *v*.
void setVersion(int v)	Sets the cookie protocol version to *v*, which willbe 0 or 1.
void doDelete(HttpServletRequest *req*, HttpServletResponse *res*) throws IOException, ServletException	Performs an HTTP DELETE.
void doGet(HttpServletRequest *req*, HttpServletResponse *res*) throws IOException, ServletException	Performs an HTTP GET.

Table-15.9. The Methods Defined by HttpServlet

The HttpSessionEvent Class

HttpSessionEvent encapsulates session events. It extents **EventObject** and is generated when a change occurs to the session. It defines this constructor:

HttpSessionEvent(HttpSession *session*)

Here, *session* is the source of the event.

Method	Description
void doHead(HttpServletRequest req, HttpServletResponse res) throws IOException, ServletException	Performs an HTTP HEAD.
void doOptions(HttpServletRequest req, HttpServletResponse res) throws IOException, ServletException	Performs an HTTP OPTIONS.
void doPost(HttpServletRequest req, HttpServletResponse res) throws IOException, ServletException	Performs an HTTP POST.
void doPut(HttpServletRequest req, HttpServletResponse res) throws IOException, ServletException	Performs an HTTP PUT.
void doTrace(HttpServletRequest req, HttpServletResponse res) throws IOException, ServletException	Performs an HTTP TRACE.
LonggetLastModified(HttpServletRequest req)	Returns the time (inmilliseconds since midnight, January 1, 1970, GMT) when the requested resource was last modified.
void service(HttpServletRequest *req*, HttpServletResponse *res*) throws IOException, ServletException	Called by the server when an HTTP request arrives for this servlet. The arguments provide access to the HTTP request and response, respectively.

Table-15.9. The Methods Defined by HttpServlet (continued)

HttpSessionEvent defines one method, **getSession( )**, which is shown here:

HttpSession getSession( )It returns the session in which the event occurred.

The HttpSessionBindingEvent Class

The **HttpSessionBindingEvent** class extends **HttpSessionEvent**. It is generatedwhen a listener is bound to or unbound from a value in an **HttpSession** object. It isalso generated when an attribute is bound or unbound.Here are its constructors:

HttpSessionBindingEvent(HttpSession *session*, String *name*)

HttpSessionBindingEvent(HttpSession *session*, String *name*, Object *val*)

Here, *session* is the source of the event and *name* is the name associated with the objectthat is being bound or unbound. If an attribute is being bound or unbound, its value ispassed in *val*.

The **getName( )** method obtains the name that is being bound or unbound. Its isshown here:
String getName( )

The **getSession( )** method, shown next, obtains the session to which the listener isbeing bound or unbound:
HttpSession getSession( )

The **getValue( )** method obtains the value of the attribute that is being bound orunbound. It is shown here:
Object getValue( )

15.7 HANDLING HTTP REQUESTS AND RESPONSES

The **HttpServlet** class provides specialized methods that handle the various types ofHTTP requests. A servlet developer typically overrides one of these methods. Thesemethods are **doDelete( )**, **doGet( )**, **doHead( )**, **doOptions( )**, **doPost( )**, **doPut**
( ), and**doTrace( )**. A complete description of the different types of HTTP requests is beyondthe scope of this book. However, the GET and POST requests are commonly usedwhen handling form input. Therefore, this section presents examples of these cases.Handling HTTP GET RequestsHere we will develop a servlet that handles an HTTP GET request. The servlet is invokedwhen a form on a Web page is submitted. The example contains two files. A Web pageis defined in **ColorGet.htm** and a servlet is defined in **ColorGetServlet.java**. The HTMLsource code for **ColorGet.htm** is shown in the following listing. It defines a form thatcontains a select element and a submit button. Notice that the action parameter of theform tag specifies

a URL. The URL identifies a servlet to process the HTTP GET request.

Listing 15.4

```
<html>
<body>
<center>
<form name="Form1"
action="http://localhost:8080/examples/servlet/ColorGetServlet">
<B>Color:</B>
<select name="color" size="1">
<option value="Red">Red</option>
<option value="Green">Green</option>
<option value="Blue">Blue</option>
</select>
<br><br>
<input type=submit value="Submit">
</form>
</body>
</html>
```

The source code for **ColorGetServlet.java** is shown in the following listing. The**doGet( )** method is overridden to process any HTTP GET requests that are sent tothis servlet. It uses the **getParameter( )** method of **HttpServletRequest** to obtain theselection that was made by the user. A response is then formulated.

Listing 15.5

```
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;
public class ColorGetServlet extends HttpServlet {
public void doGet(HttpServletRequest request,
HttpServletResponse response)
throws ServletException, IOException {
String color = request.getParameter("color");
response.setContentType("text/html");
PrintWriter pw = response.getWriter();
pw.println("<B>The selected color is: ");
pw.println(color);
pw.close();
}
}
```

Compile the servlet and perform these steps to test this example:

1. Start Tomcat, if it is not already running.
2. Display the Web page in a browser.
3. Select a color.
4. Submit the Web page.

After completing these steps, the browser will display the response that is dynamicallygenerated by the servlet.One other point: Parameters for an HTTP GET request are included as part of theURL that is sent to the Web server. Assume that the user selects the red option andsubmits the form. The URL sent from the browsertothe server ishttp://localhost:8080/examples/servlet/ColorGetServlet?color= RedThe characters to the right of the question mark are known as the *query string*.Handling HTTP POST RequestsHere we will develop a servlet that handles an HTTP POST request. The servlet isinvoked when a form on a Web page is submitted. The example contains two files. AWeb page is defined in **ColorPost.htm** and a servlet is defined in **ColorPostServlet.java**.The HTML source code for **ColorPost.htm** is shown in the following listing. It isidentical to **ColorGet.htm** except that the method parameter for the form tag explicitlyspecifies that the POST method should be used, and the action parameter for the formtag specifies a different servlet.

Listing 15.6

```
<html>
<body>
<center>
<form name="Form1"
method="post"
action="http://localhost:8080/examples/servlet/ColorPostServlet"
>
<B>Color:</B>
<select name="color" size="1">
<option value="Red">Red</option>
<option value="Green">Green</option>


<option value="Blue">Blue</option>
</select>
<br><br>
<input type=submit value="Submit">
</form>
</body>
</html>
```

The source code for **ColorPostServlet.java** is shown in the following listing. The**doPost( )** method is overridden to process any HTTP POST requests that are sent tothis servlet. It uses the **getParameter( )** method of **HttpServletRequest** to obtain theselection that was made by the user. A response is then formulated.

Listing 15.8

```
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;
public class ColorPostServlet extends HttpServlet {
public void doPost(HttpServletRequest request,
HttpServletResponse response)
throws ServletException, IOException {
String color = request.getParameter("color");
response.setContentType("text/html");
PrintWriter pw = response.getWriter();
pw.println("<B>The selected color is: ");
pw.println(color);
pw.close();
}
}
```

Compile the servlet and perform the same steps as described in the previoussection to test it.

Note: Parameters for an HTTP POST request are not included as part of the URL thatis sent to the Web server. In this example, the URL sent from the browser to the server is:http://localhost:8080/examples/servlet/ColorGetServlet

The parameter names and values are sent in the body of the HTTP request.

Using Cookies :

Now, let's develop a servlet that illustrates how to use cookies. The servlet is invokedwhen a form on a Web page is submitted. The example contains three files assummarized here:

File	Description
AddCookie.htm	Allows a user to specify a value for the cookienamed **MyCookie**.
AddCookieServlet.java	Processes the submission of **AddCookie.htm**.
GetCookiesServlet.java	Displays cookie values.

The HTML source code for **AddCookie.htm** is shown in the following listing.

This page contains a text field in which a value can be entered. There is also a submitbutton on the page. When this button is pressed, the value in the text field is sent to

AddCookieServlet via an HTTP POST request.

Listing 15.9

```
<html>
<body>
<center>
<form name="Form1"
method="post"
action="http://localhost:8080/examples/servlet/AddCookieServlet">
<B>Enter a value for MyCookie:</B>
<input type=textbox name="data" size=25 value="">
<input type=submit value="Submit">
</form>
</body>
</html>
```

The source code for **AddCookieServlet.java** is shown in the following listing. Itgets the value of the parameter named “data”. It then creates a **Cookie** object that hasthe name “MyCookie” and contains the value of the “data” parameter. The cookie isthen added to the header of the HTTP response via the **addCookie( )** method. A feedbackmessage is then written to the browser.

Listing 15.10

```
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;


public class AddCookieServlet extends HttpServlet {
public void doPost(HttpServletRequest request,
HttpServletResponse response)
throws ServletException, IOException {
// Get parameter from HTTP request.
String data = request.getParameter("data");
// Create cookie.
Cookie cookie = new Cookie("MyCookie", data);
// Add cookie to HTTP response.
response.addCookie(cookie);
// Write output to browser.
response.setContentType("text/html");
PrintWriter pw = response.getWriter();
pw.println("<B>MyCookie has been set to");
pw.println(data);
pw.close();
}
}
```

The source code for **GetCookiesServlet.java** is shown in the following listing. Itinvokes the **getCookies( )** method to read any cookies that are included in the HTTPGET request. The names and values of these cookies are then written to the HTTPresponse. Observe that the **getName( )** and **getValue( )** methods are called to obtain this information.

Listing 15.11

```
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;
public class GetCookiesServlet extends HttpServlet {
public void doGet(HttpServletRequest request,
HttpServletResponse response)
throws ServletException, IOException {
// Get cookies from header of HTTP request.
Cookie[] cookies = request.getCookies();
// Display these cookies.
response.setContentType("text/html");
PrintWriter pw = response.getWriter();
pw.println("<B>");
for(int i = 0; i < cookies.length; i++) {
String name = cookies[i].getName();
String value = cookies[i].getValue();
pw.println("name = " + name +
"; value = " + value);
}
pw.close();
}
}
```

Compile the servlet and perform these steps:

1. Start Tomcat, if it is not already running.
2. Display **AddCookie.htm** in a browser.
3. Enter a value for **MyCookie.**
4. Submit the Web page.

After completing these steps you will observe that a feedback message is displayed bythe browser.Next, request the following URL via the browser:

http://localhost:8080/examples/servlet/GetCookiesServlet

Observe that the name and value of the cookie are displayed in the browser.In this example, an expiration date is not explicitly assigned to the cookie via the**setMaxAge( )** method of **Cookie**. Therefore, the cookie expires when the browsersession ends. You can experiment by using **setMaxAge( )** and observe that the cookieis then saved to the disk on the client machine.

Session TrackingHTTP is a stateless protocol. Each request is independent of the previous one. However,in some applications, it is necessary to save state information so that information canbe collected from several interactions between a browser and a server. Sessions providesuch a mechanism.

A session can be created via the **getSession( )** method of **HttpServletRequest**. An**HttpSession** object is returned. This object can store a set of bindings that associatenames with objects. The **setAttribute( )**, **getAttribute( )**, **getAttributeNames( )**, and**removeAttribute( )** methods of **HttpSession** manage these bindings. It is importantto note that session state is shared among all the servlets that are associated with aparticular client.

The following servlet illustrates how to use session state. The **getSession( )** methodgets the current session. A new session is created if one does not already exist. The**getAttribute( )** method is called to obtain the object that is bound to the name “date”.That object is a **Date** object that encapsulates the date and time when this page was lastaccessed. (Of course, there is no such binding when the page is first accessed.) A **Date**object encapsulating the current date and time is then created. The **setAttribute( )**method is called to bind the name “date” to this object.

Listing 15.12

```
import java.io.*;
import java.util.*;
import javax.servlet.*;
import javax.servlet.http.*;
public class DateServlet extends HttpServlet {
public void doGet(HttpServletRequest request,
HttpServletResponse response)
throws ServletException, IOException {
// Get the HttpSession object.
HttpSession hs = request.getSession(true);
// Get writer.
response.setContentType("text/html");
PrintWriter pw = response.getWriter();
pw.print("<B>");
// Display date/time of last access.
Date date = (Date)hs.getAttribute("date");
if(date != null) {
pw.print("Last access: " + date + "<br>");
}
// Display current date/time.
date = new Date();
hs.setAttribute("date", date);
pw.println("Current date: " + date);
}
}
```

When you first request this servlet, the browser displays one line with the currentdate and time information. On subsequent invocations, two lines are displayed. Thefirst line shows the date and time when the servlet was last accessed. The second lineshows the current date and time.

15.8 SUMMARY

This chapter covers what is servlet. The life cycle of servlet, how servlet is executed. It covers all the classes, interfaces and methods required to execute servlet.

It covers reading parameters from HTMl files and processes it to servlet. It also covers how to store values on client machine through cookies and sessions.

15.9 QUESTION

1. Explain life cycle of servlet.
2. What are the steps to execute the servlet program?
3. Write a code to read a value from HTML and calculate the square of numbers at server end.
4. Explain cookies with an example.
5. Explain Sessions in servlet.

*****